IMPLEMENTACIÓN DE APLICACIÓN WEB DE ADMINISTACIÓN PARA PRODUCTO GESTOR DE APARCAMIENTOS EN EMPRESAS

RUBÉN PÉREZ LÓPEZ

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de Fin de Grado

20/05/2019

Director:

Rubén Fuentes Fernández

Agradecimientos

Me gustaría agradecer en primer lugar a mi familia y amigos por haberme dado su apoyo durante todos estos años de carrera.

A todos los profesores que se han esforzado por transmitir sus conocimientos y en especial a mi tutor de TFG, Rubén Fernández.

Y por último y no menos importante, me gustaría agradecer a mis compañeros de CRC [1], especialmente a Gonzalo Gimeno por ayudarme y apoyarme siempre que lo he necesitado.

Gracias

Resumen

Este proyecto es parte de un producto empresarial actualmente en desarrollo para la gestión de aparcamientos en empresas. Estos aparcamientos incluyen aquellas plazas de aparcamiento de las que pueden hacer uso los empleados. El producto facilitará la gestión de las plazas y su reserva por los empleados.

El producto está formado por dos aplicaciones principales. Una aplicación móvil para los empleados y una aplicación web para la administración. Esta última es el objetivo de este Trabajo de Fin de Grado (TFG).

La aplicación móvil ya ha sido desarrollada. Permite que los empleados de una empresa pueden reservar plazas de aparcamiento. En este desarrollo se han creado también los correspondientes servicios web con API REST.

La aplicación web de administración permite gestionar los aparcamientos y su información relacionada. Para ello, se crean los correspondientes servicios web también con API REST.

El trabajo ha sido desarrollado y desplegado usando las tecnologías empresariales del producto global. Los servicios web han sido desarrollados con Java y Spring Framework. Estos se usan desde una interfaz de usuario desarrollada con tecnologías web, concretamente con React y Redux, además de otras librerías y utilidades.

El sistema completo de gestión de aparcamientos se despliega en Google Cloud Platform, utilizando herramientas para ello como Cloud SQL o Kubernetes.

La aplicación de gestión resultante corresponde a un desarrollo industrial. Sigue pautas de desarrollo ágil y despliegue continuo para productos de alta disponibilidad en la nube.

Palabras clave

Gestión de aparcamientos, cloud, alta disponibilidad, despliegue continuo, aplicación web, servicio API REST.

Abstract

This project is part of a business product currently under development for the management of car parks in companies. These car parks include those parking spots that employees can use. The product will facilitate the management of places and their reservation by employees.

The product consists of two main applications. A mobile application for employees and a web application for administration. This last application is the objective of this Final Degree Project.

The mobile application has already been developed. It allows employees of a company to reserve parking spots. In this development, the corresponding web services with REST API have also been created.

The web administration application allows you to manage the car parks and their related information. To do this, the corresponding web services are also created with the REST API.

This Final Degree Project has been developed and deployed using the global product business technologies. The web services have been developed with Java and Spring Framework. These are used from a user interface developed with web technologies, specifically React and Redux, as well as other libraries and utilities.

The complete parking management system is deployed in Google Cloud Platform, using tools such as Cloud SQL or Kubernetes.

The resulting management application corresponds to an industrial development. It follows guidelines for agile development and continuous deployment for high availability products in the cloud.

Keywords

Parking management, cloud, high availability, continuous deployment, web application, REST API service.

Índice

Índice de figuras

Índice de tablas

Glosario

API	*Application Programming Interface*
CRC	CRC Information Technologies
CRUD	*Create, Read, Update, Delete*
CSS	*Cascading Style Sheets*
DI	*Dependency Injection*
DOM	*Document Object Model*
DTO	*Data Transfer Object*
GCP	*Google Cloud Platform*
GKE	*Google Kubernetes Engine*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IC	Integración Continua
IDE	*Integrated Development Environment*
IoC	*Inversion of Control*
IU	Interfaz de Usuario
JSON	*JavaScript Object Notation*
k8s	Kubernetes
ORM	*Object-Relational Mapping*
POO	Programación Orientada a Objetos
POM	*Project Object Model*
REST	*Representational State Transfer*
SQL	*Structured Query Language*
STS	*Spring Tool Suite*
TFG	Trabajo de Fin de Grado

UML	*Unified Modeling Language*
XML	*Extensible Markup Language*
YAML	*Yet Another Markup Language*

Capítulo 1 - Introducción

1.1 Motivación

Este Trabajo de Fin de Grado (TFG) ha sido realizado junto con la empresa CRC Information Technologies (CRC). CRC se define en su página web como una compañía española de consultoría y servicios que dirige sus prácticas de negocio a la definición e implementación de sistemas para solucionar las necesidades de los clientes, con el objetivo de proporcionarles ventajas competitivas fundamentadas en las tecnologías de la información [1].

El TFG es parte de un producto software de gestión colaborativa de aparcamientos. Este pretende facilitar la gestión y reserva de plazas de aparcamiento reservadas a los empleados.

El producto se compone de una aplicación móvil en la que los empleados pueden reservar plazas de aparcamiento y una aplicación web para la administración y gestión, todo ello desplegado en Google Cloud Platform. La aplicación móvil ya ha sido desarrollada por la empresa, siendo la aplicación web de gestión el objetivo de este trabajo.

1.2 Objetivos

El objetivo general del proyecto ha sido desarrollar e implementar una aplicación web que ayude a las empresas a gestionar sus plazas de aparcamiento. Ésta se integrará en un producto mayor actualmente en desarrollo, que también incluye una aplicación móvil para que los empleados puedan reservar plazas. El producto completo constituirá una solución empresarial para la gestión completa de plazas de aparcamiento de empresas.

Es necesario que la aplicación a desarrollar sea fácilmente escalable, modular y mantenible debido a que se prevé que el producto desarrollado crezca y evolucione con nueva funcionalidad, correcciones y actualizaciones. Mantener esta filosofía es fundamental para que los cambios venideros requieran menor tiempo y esfuerzo para su implementación e integración.

Este trabajo consiste por tanto en el desarrollo de dicha aplicación web para la administración y gestión del sistema, su integración con la aplicación de movilidad y otros servicios en Google Cloud Platform, y su despliegue en producción en un entorno empresarial de alta disponibilidad e integración continua.

1.3 Plan de trabajo

La realización del proyecto en la empresa está guiada por el uso de metodologías ágiles. La realización del trabajo individual para el TFG, al estar integrado en este proyecto, ha seguido estas pautas de realización. A continuación, se recoge el marco del desarrollo en la empresa.

El proceso realizado para desarrollar e implementar la última fase del producto tiene una duración de aproximadamente seis meses. Los integrantes del grupo son

cuatro: dos desarrolladores, entre los que se incluye el autor de este TFG, un jefe de proyecto y un cliente que determina el producto. Todos los integrantes del grupo tienen asignados otros proyectos, por lo que la dedicación del tiempo es parcial.

El desarrollo de la aplicación se hizo siguiendo la metodología ágil Scrum [2]. Esto implica un desarrollo incremental e iterativo, donde el cliente (*product owner* de Scrum) participa como un miembro más del equipo para guiar las iteraciones (*sprints*). Se decidió que las reuniones serían cada dos semanas y con ellas las iteraciones. En las reuniones se informaba de los avances logrados, dificultades encontradas y las soluciones aplicadas a dichos problemas, además de decidir y priorizar las tareas a realizar por cada miembro del equipo para la siguiente iteración. Además, se realizaban *daily scrums* , que son reuniones diarias para la sincronización del equipo de desarrollo.

Las herramientas utilizadas para gestionar el proceso han sido Microsoft Teams y Redmine. Microsoft Teams para programar reuniones y almacenar los documentos relacionados con el proceso de desarrollo. Redmine para gestionar y tener un seguimiento de las tareas que se iban definiendo.

Por otro lado, se han utilizado herramientas de gestión del código. Se han utilizado Git como servicio de control de versiones, Gitlab como gestor de repositorios y Jenkins para IC (Integración Continua).

1.4 Estructura de la memoria

En los siguientes capítulos de la memoria obtendremos la información sobre el proyecto. La estructura se compone de:

- Capítulo 2, **Estado del arte**. Discute los productos y aplicaciones que ofrece el mercado actual para gestores colaborativos de aparcamientos.
- Capítulo 3, **Requisitos**. Describe los requisitos funcionales del producto y la aplicación del TFG.
- Capítulo 4, **Tecnologías**. Explica las tecnologías y herramientas utilizadas en la aplicación.
- Capítulo 5, **Implementación**. Describe la arquitectura y estructura del proyecto de forma global y de la aplicación web de administración con mayor detalle.
- Capítulo 6, **Interfaz de usuario y funcionalidad**. Detalla las diferentes pantallas de la aplicación web de administración y su funcionalidad.
- Capítulo 7, **Arranque y despliegue de la aplicación**. Muestra cómo se realiza el arranque de la aplicación en diferentes entornos y cómo se despliega en producción.
- Capítulo 8, **Conclusiones**. Analiza los resultados del proyecto y el trabajo futuro a realizar.

Chapter 1 - Introduction

1.1 Motivation

This Final Degree Project has been carried out together with the company CRC Information Technologies (CRC). CRC is defined itself on its website as a Spanish consulting and services company that focuses its business practices on the definition and implementation of systems to solve their customers' needs, with the overall goal of providing competitive advantages based on information technologies [1].

This end-of-degree project is part of a collaborative parking-management software product. That product aims to facilitate the management and reservation of parking spaces reserved for employees.

The product consists of a mobile application in which employees can reserve parking spaces and a web application for administration and management, all deployed in Google Cloud Platform. The mobile application has already been developed by the company, and the development of the management web application is the objective of this work.

1.2 Goals

The general objective of the project has been to develop and implement a web application that helps companies manage their parking spots. This will be integrated into a larger product currently under development, which also includes a mobile application so that employees can reserve places. The complete product will constitute a business solution for the complete management of business parking spaces.

It is necessary that the application to be developed is easily scalable, modular and maintainable because it is foreseen that the developed product will grow and evolve with new functionality, corrections and updates. Maintaining this philosophy is essential so that future changes require less time and effort for its implementation and integration.

This work consists therefore in the development of this web application for the administration and management of the system, its integration with the application of mobility and other services in Google Cloud Platform, and its deployment in production in a business environment of high availability and continuous integration.

1.3 Work plan

The construction of the project in the company is guided by the use of agile methodologies. The realization of individual work for the Final Degree Project, since it is integrated into this project, has followed these guidelines. Next, the framework of the development in the company is shown.

The process carried out to develop and implement the last phase of the product lasts approximately six months. The members of the group are four: two developers, including the author of this end degree project, a project manager and a customer who determines the product. All the members of the group are assigned other projects, so the dedication of the time is partial.

The development of the application was done following the agile methodology Scrum [2] This implies an incremental and iterative development, where the client (Scrum product owner) participates as a member of the team to guide the iterations (sprints). It was decided that the meetings would be every two weeks and with them the iterations. The meetings reported on the progress made, difficulties encountered, and the solutions applied to these problems, as well as deciding and prioritizing the tasks to be performed by each team member for the next iteration. In addition, daily scrums were held, which are daily meetings for the synchronization of the development team.

The tools used to manage the process have been Microsoft Teams and Redmine. Microsoft Teams to schedule meetings and store documents related to the development process. Redmine to manage and keep track of the tasks that were being defined.

On the other hand, code management tools have been used. Git has been used as a version control service, Gitlab as a repository manager and Jenkins for CI (Continuous Integration).

1.4 Memory structure

In the following chapters of the report we will obtain the information about the project. The structure is composed of:

- Chapter 2, **State of the art.** It discusses the products and applications offered by the current market for collaborative parking managers.
- Chapter 3, **Requirements**. It describes the functional requirements of the product and the application of the Final Degree Project.
- Chapter 4, **Technologies**. It explains the technologies and tools used in the application.
- Chapter 5, **Implementation**. It describes the architecture and structure of the project in a global way and the administration web application in greater detail.In
- Chapter 6, **User interface and functionality**. It details the different screens of the administration web application and its functionality.
- Chapter 7, **Application booting up and deployment**. It shows how the application is booted up in different environments and how it is deployed in production.
- Chapter 8: **Conclusions**. It analyzes the results of the project and the future work to be done.

Capítulo 2 - Estado del arte

2.1 Aplicaciones para gestión de aparcamiento en empresas

Existen numerosos sistemas para la explotación de aparcamientos de centros comerciales y empresariales. Estos suelen enfocar el problema en la gestión y búsqueda de aparcamiento de los clientes. Implementan el software necesario y además las máquinas y el equipo para el aparcamiento, como lector de matrícula, sensores y máquinas de pago. No consideran específicamente la problemática de los aparcamientos de las empresas y los empleados.

Existen algunos ejemplos de gestores de aparcamientos para empresas. A continuación, se discuten algunos centrándose en las aplicaciones software relacionadas.

4Park Office

4Park Office es un sistema de control y gestión de aparcamientos en oficinas. Lo desarrolla la empresa i+D3 [3].

Figura 2-1. Logotipo de i+D3

Este sistema permite a las empresas gestionar sus aparcamientos y plazas disponibles, así como los empleados y los visitantes que pueden utilizar dichas plazas. Incluye el equipamiento para gestionar el acceso al aparcamiento (barrera automática, lector de matrículas, sensores de guiado, etc.) y software necesario para la gestión.

El producto que pretende desarrollar CRC se centra únicamente en la parte software, contando también con una aplicación para administrar plazas de aparcamiento y empleados. A diferencia de 4Park Office, el producto de CRC también ofrece una aplicación móvil a los empleados en la que pueden solicitar una plaza de aparcamiento y dispone de la posibilidad de sortear las plazas entre los empleados que las solicitan.

Sistema de guiado para aparcamientos de empresas y centros logísticos

La empresa Urbiotica ofrece una solución de *Sistema de guiado para aparcamientos de empresas y centros logísticos*, ofreciendo el hardware (sensores y paneles digitales informativos) y el software necesario para guiar a los conductores por los aparcamientos. Dispone de aplicaciones web y móvil para guiar a los usuarios y realizar un análisis del uso de las plazas y áreas de estacionamiento [4].

Figura 2-2. Logotipo de Urbiotica

A diferencia de esta solución, el producto de CRC se centra en ofrecer un servicio a los empleados de la empresa. Se centra en la reserva de plazas y el sorteo de las mismas cuando hay más solicitudes que plazas disponibles.

Sistema de Control y gestión de aparcamiento Sense

El sistema de Control y gestión de aparcamiento Sense está desarrollado por Equinsa Parking. Permite gestionar y realizar el control de accesos en aparcamientos de hospitales, centros comerciales, supermercados o edificios corporativos [5].

Se encargan de controlar, gestionar y mantener los aparcamientos, integrándolo con otros sistemas propios como el sistema de reconocedor de matrículas, el sistema de pago o sistemas propios del cliente.

Figura 2-4. Logotipo de aplicación Sense

Figura 2-3. Logotipo de Equinsa Parking

Una vez más, vemos cómo la solución se centra en el punto de vista de la empresa y en proporcionar un sistema integrado con diferentes componentes hardware para gestionar el aparcamiento y tener un control de accesos, pero no dispone de una solución al problema de ofrecer las plazas de aparcamiento de una empresa para el uso de sus empleados.

SKIDATA

SKIDATA ofrece varios servicios, como gestión de aparcamientos, control de accesos o generación de informes entre otros. Es una solución enfocada a grandes superficies, como centros comerciales, aeropuertos u hospitales [6].

Figura 2-5. Logotipo de SKIDATA

Ofrecen un servicio completo para explotar y gestionar los aparcamientos, desde el equipamiento hardware para realizar el pago, barreras automáticas o sensores, hasta el software para gestionar dichos aparcamientos.

Figura 2-6. Servicios queofrece SKIDATA

Esta solución tampoco resuelve el problema desde el punto de vista de los empleados de una empresa, que es justamente una las las principales funciones que ofrece el producto que CRC quiere desarrollar.

2.2 Conclusiones

En la *Tabla 2-1* podemos ver de forma más esquematizada qué aportan las diferentes aplicaciones que hemos visto anteriormente.

	4Park Office	SW de Urbiotica	SW Sense	SKIDATA	SW de CRC
Grandes superficies	✓	✓	✓	✓	✓
Control de accesos	✓	✓	✓	✓	✗
Control de pago	✓	✓	✓	✓	✗
Informes	✗	✓	✗	✓	✗
Equipamiento HW	✓	✓	✓	✓	✗
Aplicación móvil	✗	✓	✗	✗	✓
Reserva de plazas para empleados	✗	✗	✗	✗	✓
Sorteo de plazas para empleados	✗	✗	✗	✗	✓

Tabla 2-1. Comparación de aplicaciones

Con el análisis previo hemos podido comprobar que el producto que se va a desarrollar es novedoso y simple. Aunque hemos podido encontrar algunos desarrollos a medida, hemos comprobado una cierta necesidad de un producto que optimice la gestión de plazas de garaje y se enfoque en beneficiar directamente a los empleados. Las soluciones existentes permiten gestionar los aparcamientos de una empresa, pero

no hemos encontrado ninguna que permita sortear las plazas entre los empleados cada día o reservarlas automáticamente de una forma sencilla a través de una aplicación móvil.

Capítulo 3 - Requisitos

La realización del producto que se pretende realizar surge de la necesidad que tiene CRC de gestionar las plazas de aparcamiento correspondientes a sus oficinas. Incluye funcionalidades como sortearlas diariamente entre los empleados que las necesiten y reservar automáticamente una plaza.

El desarrollo del producto al nivel de la empresa se organizó en varias fases:

- **Primera fase**: hacer investigación de mercado, reuniones con otras empresas, definir usuarios de la aplicación móvil y de la aplicación de administración y especificar los requisitos funcionales de manera global y general.
- **Segunda fase**: abarca el diseño e implementación de la aplicación móvil.
- **Tercera fase**: se encomendó al autor de este TFG la tarea de diseñar e implementar la aplicación web de administración (ver sección *3.2.4 Implementación actual de la aplicación móvil*). El desarrollo de esta última fase es el objetivo a realizar en este TFG. En los siguientes capítulos veremos cómo se diseña e implementa la aplicación y qué tecnologías se utilizan para ello.

En este capítulo veremos los diferentes tipos de usuario del sistema y los requisitos funcionales, tanto de la aplicación móvil, como de la aplicación web de administración

3.1 Modelado de usuarios

Tras una investigación de mercado y reuniones con empresas que tenían problemas similares, se decidió poner en marcha el desarrollo del producto y se definieron dos tipos de usuario primario: usuario de la aplicación móvil y usuario administrador.

Los usuarios de la aplicación son los empleados que utilizan el parking para aparcar sus coches. Estos utilizarán la aplicación móvil que se ha desarrollado para reservar plazas de aparcamiento.

Por otro lado, los usuarios de administración serán los empleados que realizan la gestión de los aparcamientos y plazas de aparcamiento, la gestión de usuarios de la aplicación y la gestión de las reservas que se realizan.

3.2 Requisitos funcionales

Tras la investigación previa y teniendo los dos usuarios diferenciados, se procedió a realizar una especificación de requisitos para cada uno de los tipos de usuario.

Para entender estos requisitos, es necesario entender primero los siguientes conceptos:

- **App móvil**: es la aplicación móvil que usarán los usuarios finales del producto.
- **Web de administración**: aplicación web a la que accederán los administradores para gestionar las plazas y para otras labores propias de administración.
- **Cliente**: el producto soporta diferentes clientes, cada uno con sus propios datos.

- **Solicitud**: Cuando un usuario quiere solicitar una plaza para un día, realiza una solicitud, que entrará en el sorteo que corresponda.
- **Reserva**: si el usuario tiene el permiso de reserva, podrá reservar una plaza.
- **Pre-reserva**: concepto creado para referirse a una plaza de aparcamiento que está asignada a una persona durante un periodo de tiempo. Por ejemplo, un socio de la empresa tiene plaza asignada para siempre (periodo sin fecha de fin).
- **Usuario de la app**: son los usuarios que hacen uso de la app del móvil, es decir, los usuarios que van a aparcar sus vehículos en el aparcamiento.
- **Usuario de administración o administrador**: son los usuarios que van a hacer uso de la aplicación web de administración.
- **Parking**: hace referencia a un espacio diferenciado que contiene plazas de parking.
- **Plaza de parking**: hace referencia a una plaza concreta que se encuentra dentro de un parking.
- **Notificaciones push**: las notificaciones push son mensajes instantáneos que recibes en tu dispositivo.

Una de las funciones importantes de nuestro producto es realizar un sorteo entre los empleados que han solicitado una plaza para el parking un día determinado.

En las siguientes secciones se detallan los requisitos funcionales propios del sorteo, de la aplicación móvil y finalmente de la aplicación web de administración.

3.2.1 Requisitos funcionales del sorteo

Se deben sortear todas las plazas del cliente con cierta periodicidad configurable (cada día, cada dos días, cada semana…) siendo como mínimo cada día.

En el sistema existen dos tipos de sorteo:

- **Sorteo aleatorio**: un sorteo totalmente aleatorio entre las personas que han solicitado plaza.
- **Sorteo aleatorio con prioridades**: cada usuario de la app tendrá configurada una prioridad. Entre aquellas personas que hayan realizado una solicitud de plaza, se sortearán primero aleatoriamente las plazas entre las solicitudes de usuarios con mayor prioridad (pongamos prioridad 1). Después, tras cubrir las peticiones de prioridad 1, se sortearán aleatoriamente las plazas que queden libres entre las peticiones de usuarios con prioridad 2, y así sucesivamente.

A la hora de ejecutar un sorteo para un día concreto, los pasos en la ejecución del sorteo son los siguientes, en este orden:

1. **Crear reservas a partir de las pre-reservas**: para aquellas personas que tengan activa una pre-reserva en el día, se les creará una reserva en el día para la plaza que tienen asignada. De esta forma, si el usuario sabe que no va a utilizar la plaza, podría liberarla cancelando la reserva generada.
2. **Asignar reservas**: en este paso, se añaden las reservas ya creadas por aquellos usuarios que tengan permiso para realizar reservas desde la app móvil.
3. **Cancelar reservas por ausencias**: para que se cancele una reserva por ausencia, se deben dar tres condiciones:
    a. Que el usuario tenga una reserva en el día.

   b. Que el usuario tenga una ausencia en el día.
   c. Que el usuario tenga configurado que libera plaza en caso de ausencia.
4. **Sortear solicitudes de plaza**: en este último punto, con las plazas que quedan libres después de los puntos anteriores, se realiza el sorteo en la modalidad que se haya configurado, con las solicitudes de plaza que existan en el día en el cliente.

Existen dos casos en los que se puede producir la ejecución del sorteo:

- **Ejecución programada**. Un sorteo tiene una periodicidad de ejecución. Por ejemplo, cada día a las 22h, o cada dos días a las 17h, cada semana a las 15h (todos los lunes a las 15h se ejecuta el sorteo de plazas para los días que van desde el martes al lunes siguiente), etc. Cuando se ejecuta el sorteo, siempre se ejecuta como mínimo para el día siguiente. Es decir, si por ejemplo se define un sorteo para que se ejecute todos los días a las 22h, la ejecución programada de hoy a las 22h sorteará las plazas para el día de mañana.
- **Ejecución tras cancelación de reserva**. Si, por ejemplo, en el día de hoy, con las plazas ya sorteadas, alguien cancela una reserva, esa plaza volverá a sortearse. Con la cancelación de una reserva, sea cual sea la hora a la que haya cancelado (siempre que ya se haya ejecutado la ejecución programada del sorteo en ese día), se volverá a ejecutar el sorteo para asignar esa plaza libre, siguiendo los mismos pasos de ejecución antes mencionados.

3.2.2 Funcionalidad de la aplicación móvil

En esta sección definimos brevemente la funcionalidad de la aplicación móvil.

Un usuario de la app podrá:

- Consultar las solicitudes y reservas de plaza que tiene desde el día actual.
- Realizar solicitudes de plaza seleccionando un rango de fechas.
- Cancelar solicitudes de plaza, seleccionando aquellas solicitudes a borrar, para poder borrarlas todas las seleccionadas de una sola vez.
- Si el usuario de la app tiene permiso de reserva, este usuario podrá realizar reservas desde la app. Si un usuario tiene permiso de reserva, sólo podrá realizar reservas, y no solicitudes de plaza.
  - El usuario solo puede reservar plazas para él mismo.
  - Los campos a seleccionar para realizar una reserva son fecha y plaza de parking.
  - Si tiene reservas existentes, podrá cancelarlas seleccionando aquellas que quiera cancelar, pudiendo cancelar todas las seleccionadas de una sola vez.
- Configurar desde la app:
  - Si en caso de tener una ausencia, quiere que se le libere la plaza automáticamente.
  - Si quiere que le lleguen notificaciones push a la app.

El usuario recibirá una notificación push cuando se le cree una reserva, ya sea porque la cree él mismo desde la app al tener permiso, o porque se le haya creado al ejecutarse un sorteo (por la ejecución programada o la ejecución tras la cancelación de alguna otra reserva).

3.2.3 Implementación actual de la aplicación móvil

Se decidió desarrollar una aplicación híbrida [7] con Angular e Ionic que pudiese ser utilizada tanto en dispositivos Android como en dispositivos iOS.

Para entender por qué se tomó esta decisión, necesitamos saber qué ventajas e inconvenientes ofrece cada una de las opciones:

- **Aplicaciones nativas**
  Lo que distingue a las aplicaciones nativas de las aplicaciones web para móvil y de las aplicaciones híbridas es que están desarrolladas para dispositivos específicos. Por ejemplo, las aplicaciones de Android están escritas en Java y las de iPhone están escritas en Objective-C.

  La ventaja de elegir una aplicación nativa es que es la más rápida y óptima para la experiencia del usuario. Además, las aplicaciones nativas pueden interactuar con todas las funciones del sistema operativo del dispositivo, como el micrófono, la cámara, las listas de contactos, etc. Sin embargo, se requieren muchos más recursos para tener una aplicación multiplataforma (por ejemplo, dispositivos iPhone y Android) y mantenerla actualizada.

- **Aplicaciones web**
  Son básicamente sitios web con interactividad similar a una aplicación móvil. Se ejecutan en varios navegadores, como Safari o Chrome, y están escritas en HTML5 y Javascript. Si la aplicación que se quiere desarrollar no requiere funcionalidades complejas ni acceso a las características del sistema operativo, entonces la construcción de una aplicación web puede ser la opción menos costosa.

  El inconveniente es que las aplicaciones web son más lentas, menos intuitivas y no están disponibles en las tiendas de aplicaciones (Play Store o App Store).

- **Aplicaciones híbridas**
  Una aplicación híbrida combina elementos tanto de aplicaciones nativas como web. Las aplicaciones híbridas se pueden distribuir a través de las tiendas de aplicaciones, al igual que una aplicación nativa, y pueden incorporar características del sistema operativo. Al igual que una aplicación web, las aplicaciones híbridas también pueden usar tecnologías web.

  Las aplicaciones híbridas suelen ser más fáciles y más rápidas de desarrollar que las aplicaciones nativas. También requieren menos mantenimiento. Por otro lado, el rendimiento de una aplicación híbrida dependerá completamente del navegador del usuario. Esto significa que las aplicaciones híbridas casi nunca se ejecutarán tan rápido como se ejecuta una aplicación nativa.

  La ventaja de las aplicaciones híbridas es que se pueden compilar en una sola base, lo que permite agregar nuevas funcionalidades a diferentes versiones de la aplicación. Con las aplicaciones nativas, se debería replicar cada nueva función para cada plataforma.

Figura 3-1. Comparación apps nativas, híbridas y web

Para autenticarse en la app, se hace uso de autenticación contra el AD (Active Directory) de Azure.

3.2.4 Requisitos de la aplicación web de administración

En esta sección vamos a definir los requisitos funcionales a través de casos de uso basados en el estándar UML [8].

3.2.4.1 Autenticación

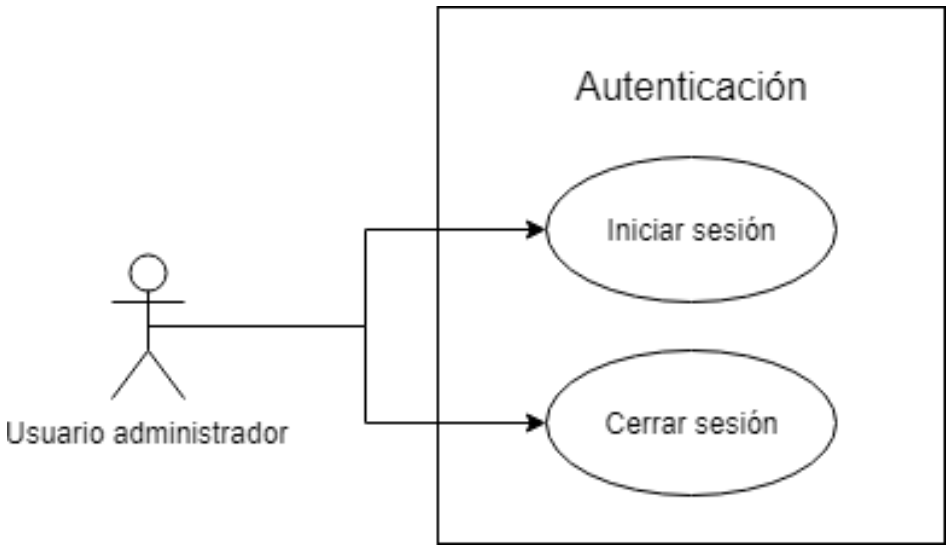

Figura 3-2. Diagrama de casos de uso de autenticación

Caso de Uso	Autenticación

Objetivo	Realizar el inicio y cierre de sesión
Actores	Usuario administrador
Disparador	Cargar la página de la aplicación
Precondiciones	• El usuario administrador no debe estar autenticado en el caso de la acción “Iniciar sesión” • El usuario administrador debe estar autenticado en el caso de la acción “Cerrar sesión”
Descripción	Habrá una página de Login para iniciar sesión en la aplicación. Los pasos a realizar son: 1) Escribir el identificador del usuario (correo electrónico) y la contraseña 2) Pulsar el botón “INICIAR SESIÓN” 3) Iniciamos sesión y accedemos a la pantalla de Parkings Para cerrar sesión habrá una opción en el menú de la aplicación para cerrar sesión.
Curso normal de los eventos	
Acción de los actores	**Respuesta del sistema**
El usuario administrador inicia sesión	El sistema registra al usuario como autenticado y le redirige a la pantalla de gestión de parkings
El usuario administrador cierra sesión	El sistema registra al usuario como no autenticado y le redirige a la pantalla de Login
Cursos alternativos	
Ninguno	

Tabla 3-1. Caso de uso Autenticación

3.2.4.2 Gestión de parkings

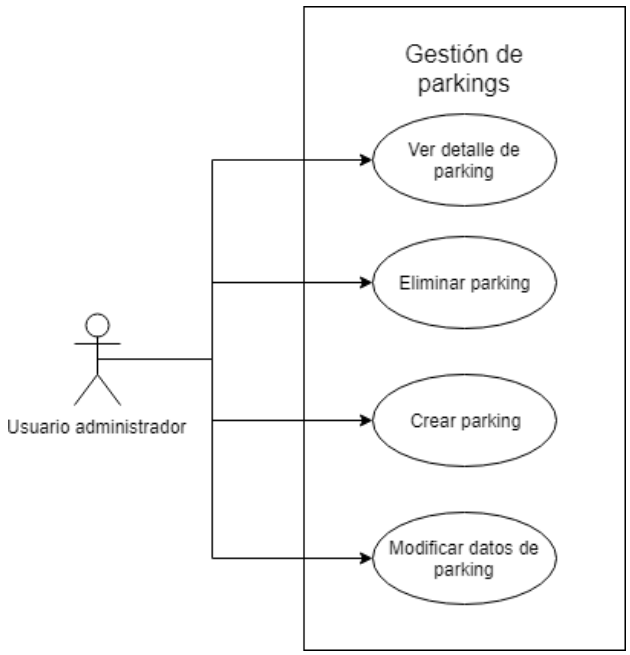

Figura 3-3. Diagrama de caso de uso de gestión de parkings

Caso de Uso	Gestión de parkings
Objetivo	Gestionar y consultar la información de los parkings
Actores	Usuario administrador
Disparador	El usuario administrador inicia sesión o selecciona la opción del menú Parkings
Precondiciones	El usuario administrador debe estar autenticado
Descripción	El administrador accede a los datos de los parkings, modificándolos o visualizándolos
Curso normal de los eventos	
Acción de los actores	**Respuesta del sistema**
El usuario administrador visualiza el detalle del parking	El sistema muestra los datos del parking
El usuario administrador modifica los datos del parking	El sistema modifica los datos del parking en base de datos
El usuario administrador elimina un parking	El sistema borra el parking de la base de datos
El usuario administrador crea un nuevo parking	El sistema inserta un nuevo parking en la base de datos
Cursos alternativos	
Ninguno	

Tabla 3-2. Caso de uso Gestión de parkings

3.2.4.3 Gestión de plazas de parking

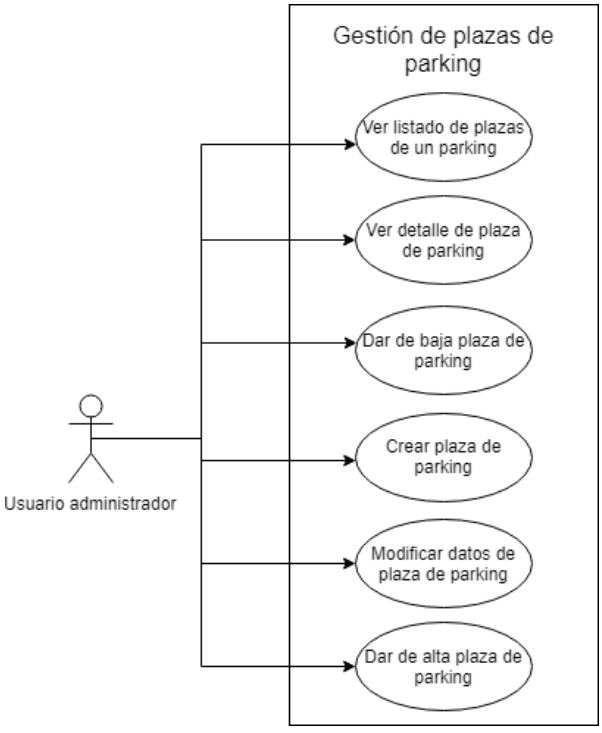

Figura 3-4. Diagrama de caso de uso de gestión plazas de parking

Caso de Uso	Gestión de plazas de parking
Objetivo	Gestionar y consultar la información de las plazas de parking
Actores	Usuario administrador
Disparador	El usuario administrador inicia sesión o selecciona la opción del menú Parkings
Precondiciones	El usuario administrador debe estar autenticado y seleccionar un parking
Descripción	El administrador accede a los datos de las plazas de parking, modificándolos o visualizándolos
Curso normal de los eventos	
Acción de los actores	**Respuesta del sistema**
El usuario administrador visualiza el listado de plazas de un parking	El sistema muestra las plazas del parking seleccionado
El usuario administrador visualiza el detalle de una plaza de parking	El sistema muestra los datos de la plaza de parking
El usuario administrador modifica los datos de la plaza de parking	El sistema modifica los datos de la plaza de parking en base de datos
El usuario administrador da de baja una plaza de parking	El sistema marca la plaza de parking como “inactiva” en la base de datos
El usuario administrador da de alta una plaza de parking inactiva	El sistema marca la plaza de parking como “activa” en la base de datos
El usuario administrador crea una nueva plaza de parking	El sistema inserta una nueva plaza de parking en la base de datos
Cursos alternativos	
Ninguno	

Tabla 3-3. Caso de uso Gestión de plazas de parking

3.2.4.4 Gestión de usuarios

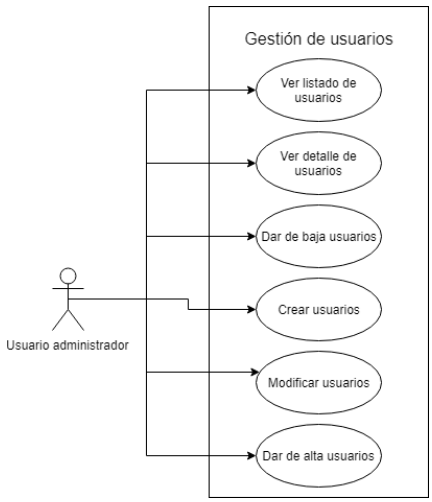

Figura 3-5. Diagrama de caso de uso de gestión de usuarios

Caso de Uso	Gestión de usuario de la app
Objetivo	Gestionar y consultar la información de los usuarios de la aplicación móvil
Actores	Usuario administrador
Disparador	El usuario administrador selecciona la opción del menú Usuarios
Precondiciones	El usuario administrador debe estar autenticado
Descripción	El administrador accede a los datos de los usuarios de la app, modificándolos o visualizándolos
Curso normal de los eventos	
Acción de los actores	**Respuesta del sistema**
El usuario administrador visualiza el listado de usuarios de la app	El sistema muestra los usuarios de la app
El usuario administrador visualiza el detalle de un usuario de la app	El sistema muestra los datos del usuario de la app
El usuario administrador modifica los datos del usuario de la app	El sistema modifica los datos del usuario de la app en base de datos
El usuario administrador da de baja un usuario de la app	El sistema marca el usuario de la app como “inactivo” en la base de datos
El usuario administrador da de alta un usuario de la app	El sistema marca el usuario de la app como “activo” en la base de datos
El usuario administrador crea un nuevo usuario de la app	El sistema inserta un nuevo usuario de la app en la base de datos
Cursos alternativos	
Ninguno	

Tabla 3-4. Caso de uso Gestión de usuarios

3.2.4.5 Gestión de reservas

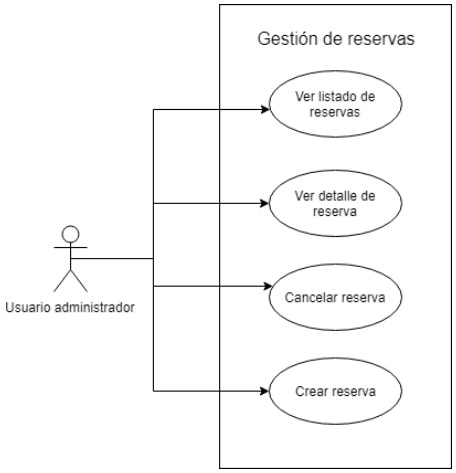

Figura 3-6. Diagrama de caso de uso de gestión de reservas

Caso de Uso	Gestión de reservas
Objetivo	Gestionar y consultar la información de las reservas de los usuarios de la app
Actores	Usuario administrador
Disparador	El usuario administrador selecciona la opción del menú Reservas
Precondiciones	El usuario administrador debe estar autenticado
Descripción	El administrador accede a los datos de las reservas los usuarios de la app, modificándolos o visualizándolos
Curso normal de los eventos	
Acción de los actores	**Respuesta del sistema**
El usuario administrador visualiza el listado de reservas de los usuarios de la app	El sistema muestra las reservas de los usuarios de la app
El usuario administrador visualiza el detalle de una reserva de un usuario de la app	El sistema muestra los datos de la reserva de usuario de la app
El usuario administrador cancela una reserva de un usuario de la app	El sistema marca la reserva como “cancelada” en la base de datos
El usuario administrador crea una nueva reserva	El sistema inserta una nueva reserva en la base de datos
Cursos alternativos	
Ninguno	

Tabla 3-5. Caso de uso Gestión de reservas

Capítulo 4 - Tecnologías

En este capítulo se detallan las tecnologías más relevantes utilizadas para llevar a cabo el desarrollo y la implementación de la aplicación web de administración.

4.1 Servicios en la nube

4.1.1 Google Cloud Platform (GCP)

El producto se despliega en la nube, concretamente en Google Cloud Platform. Google Cloud Platform es una *suite* con diferentes servicios que funcionan en la misma infraestructura que utiliza Google de manera interna. Por ejemplo, es la usada con servicios como YouTube o Google Search.

Se hace uso de Google Kubernetes Engine para desplegar los servicios y Cloud SQL como servicio de base de datos.

Cloud SQL se utiliza para configurar, mantener y administrar la base de datos MySQL de la aplicación en la nube.

Google Kubernetes Engine es un proyecto de Google creado para realizar la gestión de aplicaciones en contenedores. Permite acciones como programar el despliegue, escalar y monitorizar los contenedores, entre otras. Para ello, utiliza Kubernetes (ver sección *4.1.2*).

4.1.2 Kubernetes

Kubernetes (o k8s) es un sistema de gestión y orquestación de sistemas distribuidos mediante contenedores. Fue originalmente diseñado por Google, y donado a la Cloud Native Computing Foundation [10].

En Kubernetes se manejan una serie de conceptos que es necesario entender:

- ***Cluster***: agrupa un conjunto de máquinas físicas o virtuales, donde desplegaremos nuestra solución.
  Un cluster de Kubernetes se compone de *masters* y *nodes*. Estos pueden ser máquinas virtuales, máquinas físicas o instancias en *cloud*.
  El *Master* es una colección de servicios necesarios para gestionar k8s, y que pertenecen a su infraestructura.
  Por otro lado, un *Node* es la máquina (física o virtual) donde se va a ejecutar el trabajo que le asigne un *Master*.
- ***Pod***: es la unidad atómica de despliegue. Es decir, no se puede desplegar algo menor que un *pod*. Dentro de un *pod* pueden ejecutarse uno o varios contenedores. Un *Pod* se encontrará dentro de un *Node*.
- ***Deployment***: los *pods* se suelen desplegar dentro de un objeto más general llamado *deployment*, que añade funcionalidades como escalado y versionado.
- ***Service***: los *pods* pueden crearse y destruirse. No se puede confiar en una IP de un *pod*, porque este puede ser eliminado y recreado, o si se escalan *pods*, los nuevos generados tendrán nuevas IPs. Los *services* se encargarán de proveer de una "puerta de entrada" para acceder a los *pods*. Un *service* expondrá una IP y puerto, redirigiendo el tráfico a los *pods*.

La forma de conectar un *pod* a un *service* es mediante *labels*. En un *service* se definirán unas *labels*, y aquellos *pods* que tengan esas *labels* definidas, recibirán tráfico de este *service*.

- ***Ingress***: se encarga de exponer rutas HTTP y HTTPS desde fuera del *cluster* a los *services* del *cluster*. Es decir, es la puerta de salida de la infraestructura de k8s a Internet.

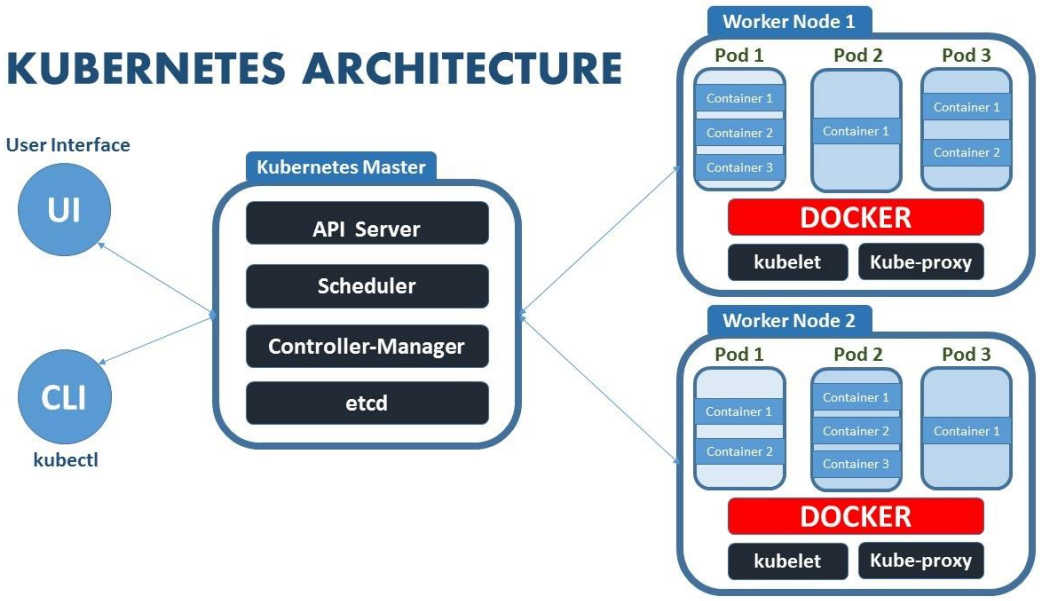

Figura 4-1. Arquitectura de Kubernetes [11]

Una aplicación puede ser desplegada en Kubernetes de la siguiente forma:

1. Se empaqueta la aplicación de forma que pueda ser arrancada en un contenedor (por ejemplo, en Docker).
2. Definir un *Pod* para ejecutar la aplicación en un contenedor.
3. Desplegar la aplicación mediante un fichero donde se presente de forma declarativa el despliegue. En estos casos se suele utilizar un fichero de *Deployment*, que ofrece opciones de escalar el despliegue. Este fichero de *Deployment* se define en un fichero YAML.

Una de las ventajas de usar Kubernetes es la forma declarativa de definir la infraestructura, servicios y despliegues, y el concepto de “estado deseado”. Lo podemos definir con el siguiente ejemplo:

1. Declaramos el estado deseado de la aplicación en un fichero.
2. Llevamos esta definición en el *Master* de Kubernetes.
3. Kubernetes almacenará esta información para definir el “estado deseado”.
4. Kubernetes implementará este “estado deseado” en el *cluster*.
5. Kubernetes hará uso de estrategias de observación mediante *loops* para asegurar que el estado actual de la aplicación coincida con el “estado deseado”.

En nuestro proyecto se hará uso de *deployments* para desplegar el backend de nuestros servicios y APIs, y *services* para exponer estas APIs. Por otro lado, también se hace uso de un *ingress* para abrir la aplicación a Internet.

Toda esta infraestructura se configura mediante código (excepto la creación del *cluster*). De esta forma, se obtiene una infraestructura fácilmente repetible, y versionada mediante código.

4.1.3 Docker

Docker es un proyecto de código abierto cuyo objetivo es crear contenedores. Estos permiten que las aplicaciones software puedan ejecutarse en cualquier ordenador que tenga Docker instalado, independientemente del sistema operativo que tenga. De esta forma se facilitan los despliegues de dichas aplicaciones.

Podemos entender un contenedor como una “caja” portable en la que podemos instalar una serie de aplicaciones software que necesitamos para ejecutar nuestra aplicación (Java, Maven, etc.).

Docker, nos permite meter en un contenedor todos aquellos elementos que necesita nuestra aplicación para ser ejecutada correctamente, además de la propia aplicación. Así nos podemos llevar ese contenedor a cualquier máquina que tenga instalado Docker y ejecutar la aplicación sin tener que preocuparnos por las versiones de software que tiene instalada esa máquina o si son compatibles.

4.2 Backend

Debido a que el *core* del sistema (donde se encuentran servicios y entidades comunes a la aplicación móvil y a la aplicación web de administración) y la aplicación móvil fueron desarrollados en Java 8, se decidió realizar también en Java 8 la aplicación web de administración para mantener la coherencia y evitar conflictos con las versiones de Java.

Como entorno de desarrollo integrado, en inglés *Integrated Development Environment* (IDE), hemos utilizado *Spring Tool Suite* (STS). Este es el IDE que CRC utiliza para sus desarrollos en Java.

4.2.1 Liquibase

Utilizamos Liquibase para administrar y aplicar cambios de esquema de base de datos. Liquibase es una librería que nos permite administrar y aplicar cambios de esquema de base de datos de forma sencilla, lo que nos resulta muy útil, especialmente en un entorno de desarrollo de software ágil. Además, nos permite poder cambiar fácilmente la base de datos o tener distintas bases de datos para los diferentes entornos.

Además, Liquibase nos permite mantener un histórico de los cambios efectuados en la base de datos. También poder realizar despliegues automáticos, ya que, en el propio despliegue, Liquibase aplicará los cambios necesarios en la base de datos, como parte del proceso de despliegue.

4.2.2 Hibernate

También utilizamos Hibernate, que es una herramienta de mapeo objeto-relacional (ORM por sus siglas en inglés) para Java. Este tipo de herramientas facilita el mapeo de atributos entre una base de datos relacional y el modelo de entidades la aplicación,

mediante archivos declarativos (XML) o anotaciones en las entidades que permiten establecer estas relaciones.

Hibernate tiene un módulo llamado Hibernate Envers, que nos permite realizar una auditoría de nuestras entidades de la aplicación, entre otras funciones.

De forma resumida, al marcar una entidad con la anotación '@Audited(targetAuditMode=RelationTargetAuditMode.NOT_AUDITED)', y teniendo Envers configurado, se crea una tabla *_*aud* sobre la entidad, donde se guarda una auditoría de cambios.

Al utilizar Envers, se crea también una única tabla *revinfo*. Esta tabla tiene dos columnas: 'REV' (que es un identificador único), y 'REVTSTMP', que es un *timestamp* del momento en el que se produjo un cambio en un registro.

Para ilustrarlo con un ejemplo, en la tabla *tpkn_reservations_aud* (ver diagrama ER), podemos observar que hay una columna 'REV'. Esta columna hace referencia al 'REV' de la tabla *revinfo*. De esta forma, mirando la referencia en *revinfo,* podemos saber el *timestamp* y cuándo se cambió el registro auditado.

Además, la columna 'REVTYPE' en las tablas *_*aud* hace referencia a la acción que se hizo sobre el registro, si fue una creación, actualización o borrado.

4.2.3 Spring

Spring es el *framework* de desarrollo de aplicaciones más popular para Java. Millones de desarrolladores en todo el mundo utilizan Spring Framework para crear código de alto rendimiento, fácil de probar y reutilizable.

Spring Framework es una plataforma Java de código abierto. Fue escrito inicialmente por Rod Johnson y se lanzó por primera vez bajo la licencia Apache 2.0 en junio de 2003.

Dos conceptos importantes de Spring que vamos a explicar son Inversión de Control e Inyección de Dependencias:

- **Inversión de Control (IoC, por sus siglas en inglés)**. Es un principio de ingeniería de software mediante el cual el control de objetos o partes de un programa se transfiere a un contenedor o marco. Se utiliza frecuentemente en la programación orientada a objetos (POO).
  A diferencia de la programación tradicional, en la que nuestro código realiza llamadas a una librería, IoC permite que un *framework* controle el flujo de un programa y realice llamadas a nuestro código. Las ventajas de esta arquitectura son:
  - Desacoplar la ejecución de una tarea de su implementación.
  - Facilitar el cambio entre diferentes implementaciones.
  - Mayor modularidad del programa.

La inversión de control se puede lograr a través de varios mecanismos, como la inyección de dependencias (DI).

- **Inyección de Dependencias (DI, por sus siglas en inglés)**. La inyección de dependencia es un patrón mediante el cual se implementa IoC, donde el control que se invierte es la configuración de las dependencias del objeto.
  El acto de conectar objetos con otros objetos, o de "inyectar" objetos en otros objetos, se realiza mediante un ensamblador en lugar de hacerlo por los propios objetos.

También utilizamos Spring Boot, que es un módulo de Spring que nos permite crear aplicaciones con Spring de forma sencilla.

Spring Boot nos permite hacer uso de Spring, pero sin la necesidad de realizar configuraciones innecesarias, basándose en convenciones. Es lo que se conoce como “convención sobre configuración”.

Spring Data es otro módulo que se encuentra dentro de la plataforma de Spring. Su objetivo es simplificar al desarrollador la persistencia de datos contra distintos repositorios de información.

4.2.4 Maven

Maven es una herramienta de gestión de proyectos. Se utiliza para la construcción, dependencia y documentación de proyectos. En pocas palabras, podemos decir que maven es una herramienta que se puede utilizar para construir y administrar cualquier proyecto basado en Java. Además, facilita el trabajo diario de los desarrolladores de Java y, en general, ayuda en la comprensión de cualquier proyecto basado en Java.

Maven usa archivos POM (Project Object Model). Estos son archivos XML que contienen información relacionada con el proyecto y la información de configuración, como las dependencias, el directorio de origen, los *plugins*, etc. utilizados por Maven para construir el proyecto. Maven lee el archivo pom.xml para llevar a cabo su configuración y operaciones. La siguiente imagen muestra la estructura típica de un fichero POM:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.sonatype.mavenbook.simple</groupId>
    <artifactId>simple</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>simple</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Figura 4-2. Ejemplo de fichero POM.xml

4.2.5 Jenkins

Jenkins es una herramienta de automatización de código abierto escrita en Java con *plugins* para Integración Continua (IC). Jenkins se utiliza para crear y probar proyectos software de forma continua, lo que facilita que los desarrolladores integren los cambios en el proyecto. También permite entregar continuamente software al integrarse con una gran cantidad de tecnologías de *testing* e implementación.

Con Jenkins, las empresas pueden acelerar el proceso de desarrollo de software a través de la automatización. Jenkins integra procesos de ciclo de vida de desarrollo de todo tipo, incluyendo compilación, documentación, pruebas, despliegue, etc.

Jenkins logra una IC con la ayuda de *plugins*. Si desea integrar una herramienta en particular, necesita instalar los *plugins* para esa herramienta, como por ejemplo Git o Maven.

La Integración Continua es una práctica de desarrollo que requiere que los desarrolladores integren el código en un repositorio compartido varias veces al día. Cada *commit* se verifica mediante una compilación automatizada, lo que permite a los equipos detectar problemas en una etapa temprana del desarrollo.

Al integrarse regularmente, se pueden detectar los errores rápidamente y localizarlos con mayor facilidad.

4.3 Frontend

Nuestra aplicación web está desarrollada con tecnologías web (HTML, CSS y Javascript) y se utilizan dos librerías de Javascript: React y Redux. En el caso de React, también se usa la librería Javascript *material-ui,* que proporciona Componentes de React y estilos basados en Material Design [22]. Material Design es un concepto, una filosofía, unas pautas enfocadas al diseño utilizado en Android, pero también en la web y en cualquier plataforma.

Otras tecnologías importantes para el desarrollo del frontend son YARN, que es la herramienta que utilizamos como gestor de dependencias.

4.3.1 React

React es una librería de JavaScript (no es un *framework*) que crea interfaces de usuario utilizando código declarativo.

Para mostrar qué es el código declarativo, vamos a utilizar el siguiente fragmento de código como ejemplo, en el que cada línea de código **declara** qué es cada elemento de la aplicación:

```
<App>
  <NavBar />
  <Header title=”Ejemplo” />
  <BookFilter />
  <BookList />
  <Footer />
</App>
```

El código declarativo describe lo que queremos en lugar de decir cómo hacerlo, como lo haría con el código imperativo. En su esencia, el código declarativo describe el resultado final, pero no actúa como una guía paso a paso de cómo hacerlo. En la práctica, eso significa que el código declarativo es más fácil de entender y cambiar, y tiene menos errores.

La forma en que mostramos la información en React es a través de componentes. Los componentes son interfaces de usuario reutilizables que dividen la aplicación en bloques separados que actúan de forma independiente. Los componentes aceptan una entrada arbitraria con datos (*prop*) y devuelven un elemento de React para declarar lo que debería aparecer en la pantalla. Pueden interactuar con otros componentes a través de *'props'* para crear una interfaz de usuario compleja.

Pero para crear una interfaz de usuario compleja, se debe ordenar los componentes de una manera lógica. Para ello, se necesita ver qué es el *'state'* o 'estado' de React.

El *'state'* es la representación de la aplicación en cualquier momento. Esto significa que, en una IU declarativa, los desarrolladores no están a cargo de cambiar la IU cuando ocurre algo. No tienen que preocuparse de ocultar o mostrar *divs*, como lo haría con una interfaz de usuario con código imperativo. Solo tenemos que preocuparnos de recibir *'state'* específico y mostrarla en la interfaz de usuario.

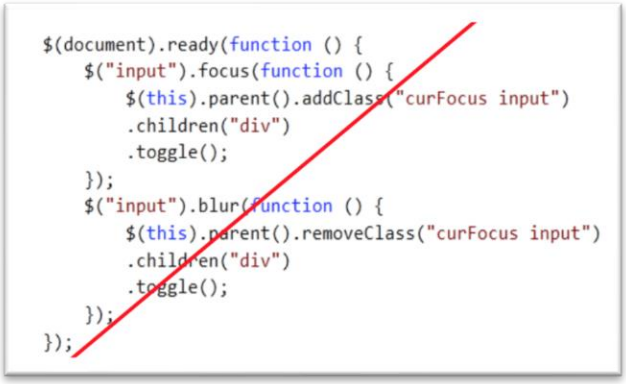

Figura 4-3. Ejemplo de código imperativo

Por lo tanto, los componentes están formados por dos conceptos principales: *'state'* y *'props'*. Gracias a ellos, podemos organizar nuestros componentes en una estructura jerárquica que garantice un flujo de datos unidireccional (a través de *'props'*), por lo que solo tenemos un estado.

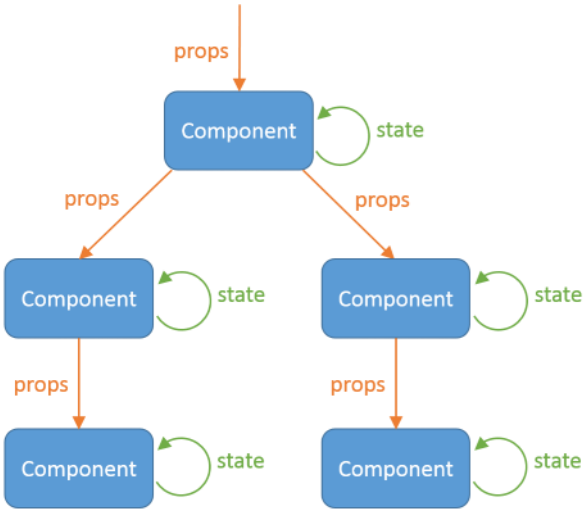

Figura 4-4. Diagrama de flujo de los componentes de React

Llamamos a esta estructura árbol, y nos permite asignar la responsabilidad de un estado a un componente. En el ejemplo siguiente, el componente 'Books' se hará cargo del *array* de libros y pasará la información a través de una *'prop'* a sus componentes hijos solo cuando lo necesiten. Los componentes hijos no pueden actualizar los datos que reciben de su padre; si los datos necesitan actualizarse, los hijos reciben otra *'prop'* de su padre con una función para actualizarlos.

El árbol de componentes nos permite crear interfaces de usuario complejas que no confunden estados. Por ejemplo, la aplicación en nuestro ejemplo no confundirá el estado de los libros con el estado de los autores, lo que podría proporcionar información contradictoria. En cambio, nuestra aplicación sigue una ruta lógica para mostrar la información correcta y la actualizará en consecuencia.

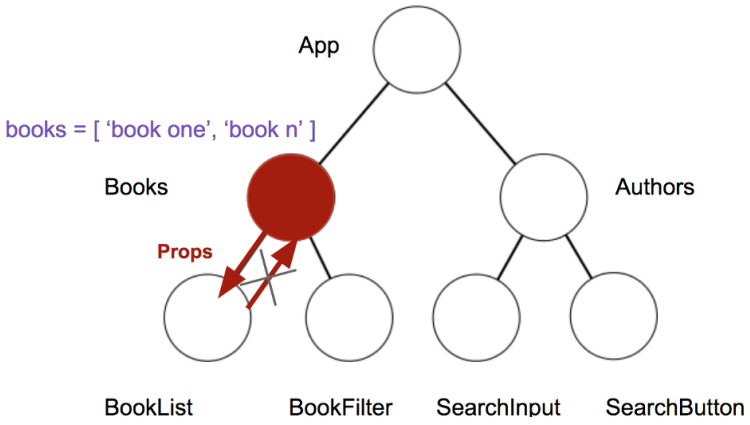

Figura 4-5. Estructura en árbol de React

Otro de los conceptos importantes de React es *Virtual DOM* o DOM virtual.

El DOM (*Document Object Model*) representa la interfaz de usuario de la aplicación. Cada vez que hay un cambio en el estado de la interfaz de usuario, el DOM se actualiza para representar ese cambio. El problema es que manipular con frecuencia el DOM afecta al rendimiento, haciéndolo lento.

El DOM se representa como una estructura de datos de árbol. Debido a eso, los cambios y actualizaciones del DOM son rápidos. Pero después de un cambio, el elemento actualizado y sus hijos deben volver a representarse para actualizar la interfaz de usuario de la aplicación. Volver a repintar la interfaz de usuario es lo que lo hace lento. Por lo tanto, cuantos más componentes de IU hay, más costosas serán las actualizaciones del DOM, ya que tienen que volver a representarse para cada actualización de DOM.

Aquí es donde el concepto de DOM virtual aparece y se tiene un rendimiento mucho mejor que el DOM real. El DOM virtual es solo una representación virtual del DOM. Cada vez que cambia el estado de nuestra aplicación, el DOM virtual se actualiza en lugar del DOM real.

Cuando se agregan nuevos elementos a la interfaz de usuario, se crea un DOM virtual, que se representa como un árbol. Cada elemento es un nodo en este árbol. Si el estado de cualquiera de estos elementos cambia, se crea un nuevo árbol virtual. Este árbol se compara con el árbol virtual anterior.

Una vez hecho esto, el DOM virtual calcula el mejor método posible para realizar estos cambios en el DOM real. Esto asegura que haya operaciones mínimas en el DOM real, reduciendo el costo de rendimiento de su actualización.

Los desarrolladores de React se abstraen de todo este proceso. Todo lo que se necesita hacer es actualizar los estados de los componentes cuando sea necesario y React se encarga del resto.

4.3.2 Redux

La página oficial define Redux de la siguiente manera [27]:

> *'Redux es un contenedor de estado predecible para aplicaciones de JavaScript.'*

Redux es una biblioteca de gestión de estado que puede conectar con cualquier librería de JavaScript, y no solo React. Sin embargo, funciona muy bien con React debido a la naturaleza funcional de React.

Como hemos visto en la sección anterior, cada componente de React tiene un estado local al que se puede acceder desde dentro del componente, o pueden pasarse a los componentes hijos como *props*, pero no es posible pasar datos desde los componentes hijos a los padres.

Además, almacenar datos de aplicaciones en el estado de un componente está bien cuando tienes una aplicación de React con pocos componentes. Sin embargo, cuando aumenta el número de niveles en la jerarquía de componentes, la administración del estado se vuelve problemática, por lo que tener un estado global de la aplicación al que todos los componentes pueden acceder, facilitaría mucho las cosas al desarrollador. Este es el gran problema al cual Redux nos ofrece la solución. Para entender cómo lo hace, se muestra en la imagen la arquitectura de Redux y se explica más adelante:

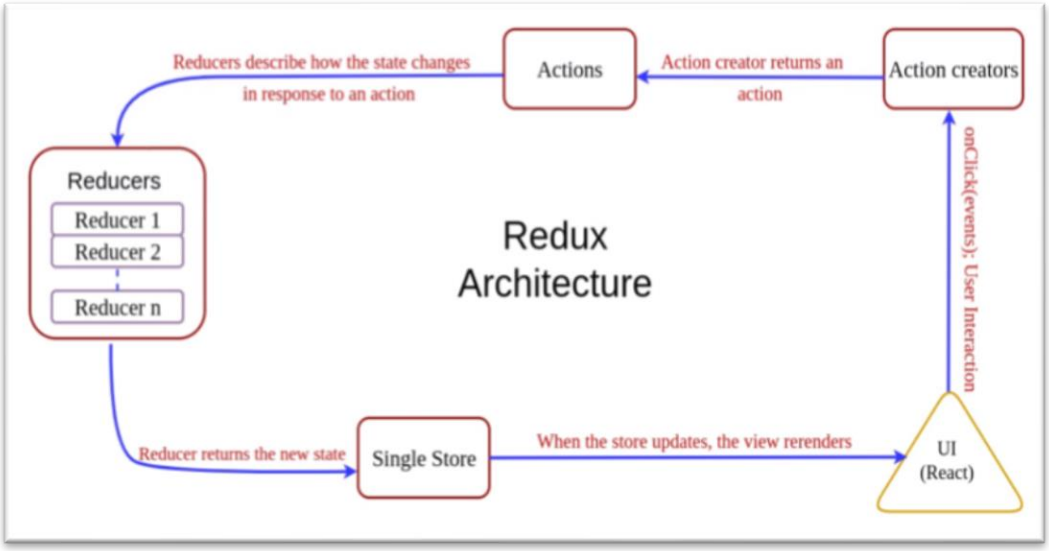

Figura 4-6. Arquitectura de Redux [28]

En Redux hay 3 principios fundamentales:

1. Redux es un contenedor de estado que almacena el estado de la aplicación en un solo lugar, al que llamaremos *store.*
2. El estado es de solo lectura, la única forma de cambiar de estado es enviar un *action*.
3. El estado sólo puede ser cambiado por “funciones puras” [29]. De forma resumida, una función pura es aquella que ante una misma entrada siempre retorna una misma salida, que no depende de ninguna variable exterior y no tiene efectos secundarios. En Redux, estas funciones puras toman el nombre de *reducers*. Los *reducers* de Redux reciben el estado anterior y un *action* y devuelven el siguiente estado.

Un *action* es básicamente un objeto que tiene una propiedad llamada *type*. También puede tener otras propiedades, pero en el siguiente ejemplo, por simplicidad sólo tendrá tipo.

```
const someAction = {type:'doSomething'}
```

Un *reducer* es simplemente una función:

```
const reducer = (state, action) => {
```

```
        if (action.type === 'doSomething'){
            return changedState;
        } else if ( action.type === 'doSomethingElse'){
            return changedState;
        } else {
            return state
        }
    }
```

El *action* que pasamos a un *reducer* determinará cómo se cambiará el estado según la propiedad *type*.

La forma que tiene React de encajar con Redux es diferenciando los componentes de React en dos tipos: *presentational components* o componentes de presentación y *container components* o componentes de contenedor [24]. Este enfoque hace que la aplicación sea más fácil de entender y permite reutilizar más fácilmente los componentes. En la siguiente tabla podemos ver un resumen de las diferencias entre *presentational components* y *container components*:

	Presentational Components	***Container Components***
Propósito	Cómo se ven las cosas (formato, estilos)	Cómo funcionan las cosas (obtener datos, actualizaciones de estado)
Son conscientes de Redux	No	Sí
Lectura de datos	Leen los datos a través de *props*	Se suscriben al *state* de Redux
Modificación de datos	Invocan funciones *callback* a través de *props*	Envían *actions* de Redux
Son escritos	A mano	Normalmente son generados con ayuda de una librería llamada React Redux

Tabla 4-1. Comparación entre Componentes de presentación y Componentes de contenedor

Capítulo 5 - Implementación

En este capítulo se explicará las diferentes partes que componen la arquitectura y estructura del sistema.

5.1 Arquitectura

En este punto se explica la estructura principal del proyecto, su arquitectura y sus principales características. A continuación, veremos cómo encaja la aplicación web de administración, que es el objetivo de este TFG, con el resto de componentes del sistema.

5.1.1 Arquitectura en entorno de producción

A continuación, se explica brevemente la arquitectura de todo el sistema:

- Arquitectura en Google Cloud Platform:
  - Google Kubernetes Engine: aquí se encuentran desplegados los diferentes backends, así como el frontend de parking-admin, que se encuentra empaquetado conjuntamente con su backend.
  - Cloud SQL: la base de datos del producto.
  - Container Registry: registro de imágenes de Docker generadas para cada servicio/aplicación.
- Arquitectura en Firebase:
  - Firebase Cloud Messaging: envío de notificaciones push a la aplicación móvil.
  - Firestore: base de datos donde se lleva un registro de los usuarios/dispositivos conectados.
- Nexus: propiedad de CRC, donde se almacenan las imágenes de Docker generadas para cada servicio/aplicación, así como las librerías propias utilizadas en el producto.

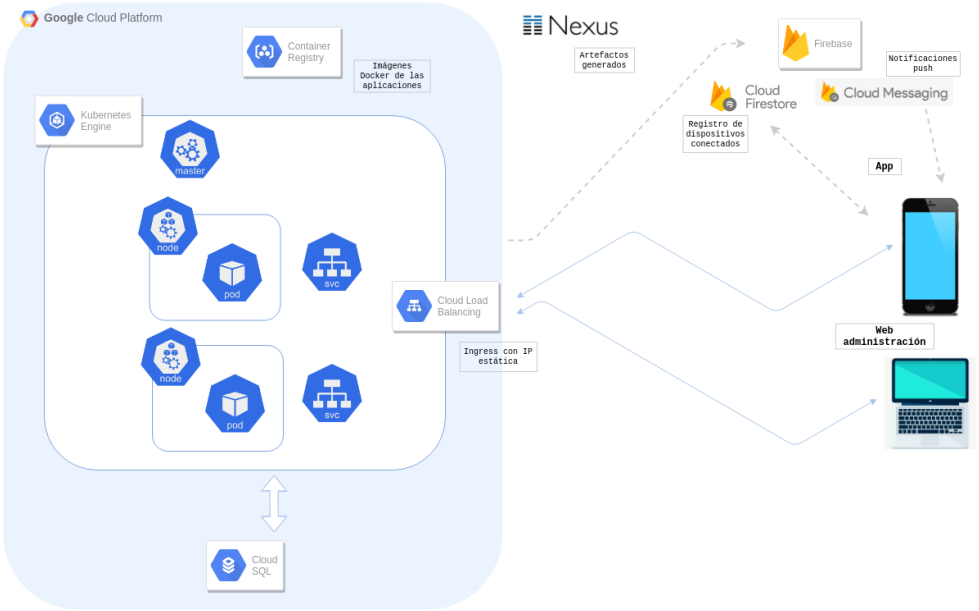

Figura 5-1. Arquitectura en entorno de producción

Por otro lado, desde otra visión, como paquetes desplegados tenemos:

- *parking-backend*: API de la aplicación móvil.
- *parking-mobile-frontend*: APK (Android) e IPA (iOS) con la aplicación móvil.
- *parking-admin*: API de la aplicación web de administración e interfaz web *parking-admin-frontend*, ambos desplegados en el mismo paquete.

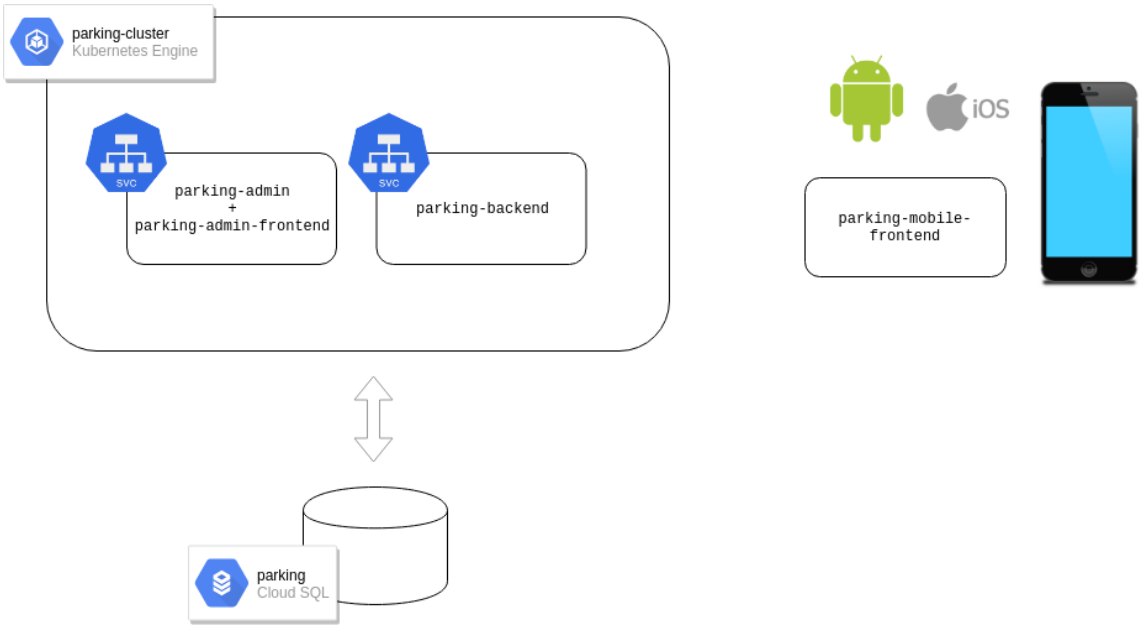

Figura 5-2. Paquetes desplegados en producción

5.1.2 Arquitectura en entorno local

En el proyecto, tenemos varios módulos de Maven:

- ***parking-core*:** librería que contiene las entidades de negocio, así como ciertos servicios y lógica común para las diferentes aplicaciones, como la ejecución de sorteos o la lógica de cancelación de reservas. Este módulo no es una aplicación ni una API. Se trata de una librería de clases que serán importadas por las diferentes aplicaciones, de tal forma que los servicios existentes en *parking-core* con su lógica asociada sean detectados por los contextos de cada aplicación y arrancados en cada aplicación. En *parking-core* se gestiona también el histórico de cambios de Liquibase, por ser el módulo donde se encuentran las entidades. La base de datos H2 en memoria es para poder generar el *changelog* de Liquibase, ya que hace falta una base de datos sobre la que realizar la comparación.
- ***parking-backend*:** backend que expone una API para ser consumida por la aplicación móvil. En local, la base de datos será una propia, una H2 en memoria.
- ***parking-mobile-frontend*:** código de la aplicación móvil.
- ***parking-admin*:** backend que expone una API para ser consumida por la aplicación web de administración. En local, la base de datos será una propia, una H2 en memoria.

- ***parking-admin-frontend*:** código de la aplicación web de administración. A la hora de construir, este frontend se empaqueta conjuntamente con *parking-admin*.

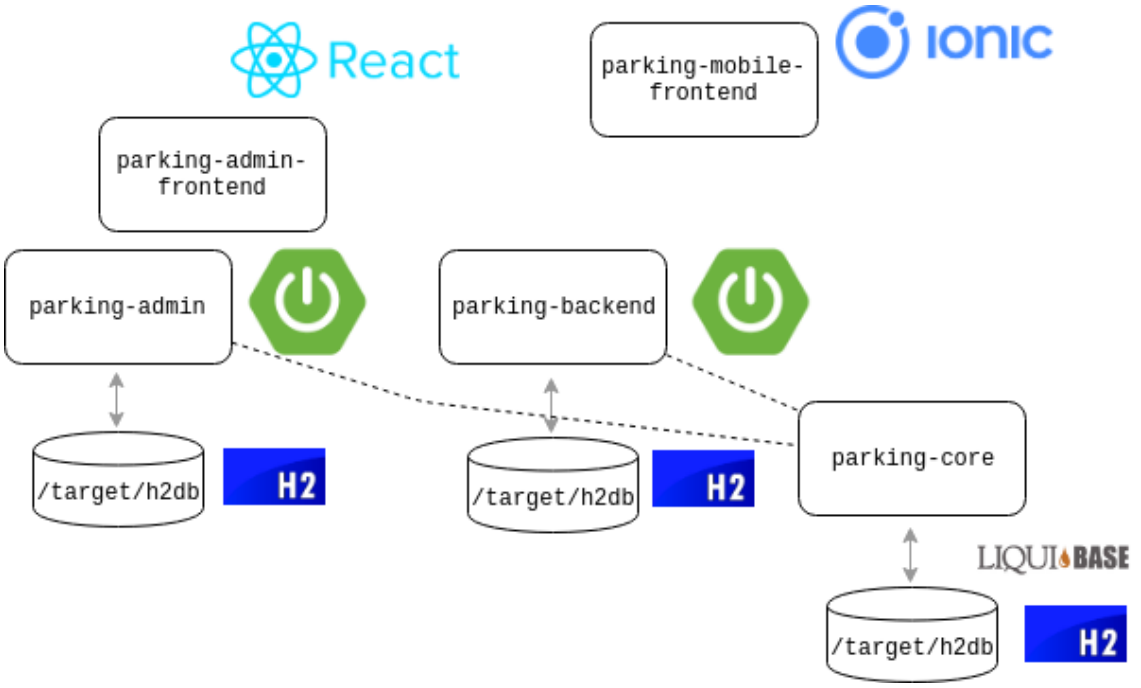

Figura 5-3. Paquetes desplegados en entorno local

Los paquetes *parking-admin-frontend* y *parking-admin* son los paquetes que se han implementado en este TFG y se explican detalladamente en las secciones *5.2* y *5.3* respectivamente.

5.1.3 Infraestructura de Integración Continua

La Integración Continua (IC) está soportada por un conjunto de pipelines. Los pipelines permiten definir el ciclo de vida completo de una aplicación (compilar, *testing*, despliegue, etc.) mediante código.

Cada módulo Maven del producto tiene dentro una carpeta */pipelines*, donde se encuentran los archivos *Jenkinsfile* con la definición de los *pipelines* correspondientes al módulo.

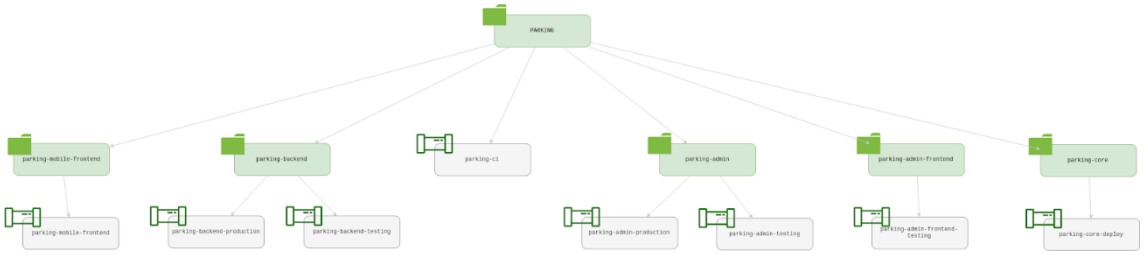

Figura 5-4. Arquitectura de IC

A continuación, se explican brevemente los *pipelines* del sistema que se muestran en la *Figura 5-4*:

- *parking-core-deploy*: sube al Nexus una nueva versión de la librería de parking-core.
- *parking-mobile-frontend*: ejecuta los tests de la aplicación móvil. La ejecución es manual, y se puede especificar la rama de Git a ejecutar.
- *parking-backend-testing*: ejecuta los tests unitarios y de integración de parking-backend. La ejecución es manual, y se puede especificar la rama de Git a ejecutar.
- *parking-backend-production*: despliega parking-backend en Producción. La ejecución es manual, y se puede especificar la rama de Git a ejecutar.
- *parking-ci*: este pipeline se ejecuta automáticamente ante cualquier cambio en la rama *develop* de Git. Este pipeline ejecutará los diferentes pipelines de test definidos anteriormente.

Además, se han programado los siguientes *pipelines* para la aplicación web de administración desarrollada en este TFG:

- *parking-admin-frontend-testing*: ejecuta los tests de la aplicación web. La ejecución es manual, y se puede especificar la rama de Git a ejecutar.
- *parking-admin-testing*: ejecuta los tests unitarios y de integración de parking-admin. La ejecución es manual, y se puede especificar la rama de Git a ejecutar.
- *parking-admin-production*: despliega parking-admin en producción. La ejecución es manual, y se puede especificar la rama de Git a ejecutar.

Para ejecutar los pasos de los pipelines, se hace uso de una imagen de Docker que tiene el JDK8 de Java, la instalación del SDK de Google Cloud y Docker instalados.

5.2 Estructura en Frontend

El frontend de la aplicación de administración se encuentra en el proyecto *parking-admin-frontend* y es la interfaz de usuario que utiliza la API que el backend nos ofrece.

La siguiente imagen muestra de forma sencilla el flujo que sigue una aplicación desarrollada con React y Redux, como se vio anteriormente en la sección *4.3 Frontend*.

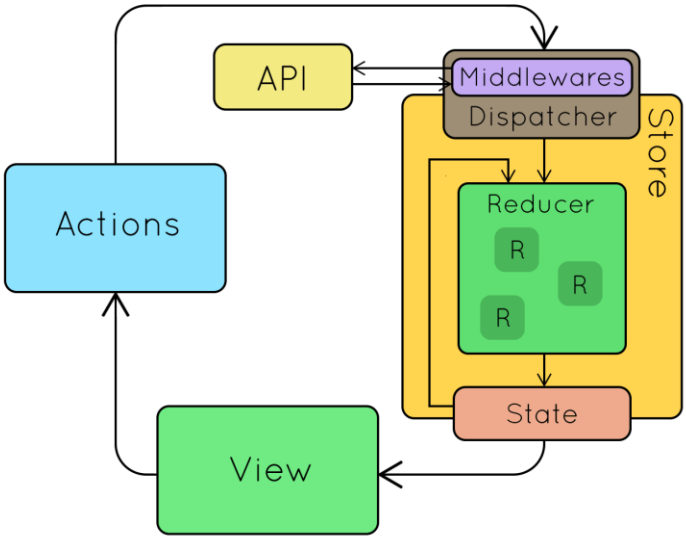

Figura 5-5. Estructura de aplicación con React y Redux

El flujo sería el siguiente:

1. El usuario lanza un evento a través de la interfaz de usuario (por ejemplo, pulsar un botón).
2. Ese evento producirá un *action* de Redux.
3. Ese *action* será enviado al *reducer* correspondiente, pero en algunas, será procesado por un *middleware* para realizar la llamada a la API del backend de la aplicación si es necesario.
   Un *middleware* es un código que se ejecuta entre el framework que recibe una solicitud, y el framework que genera una respuesta.
4. El *reducer* recibe el *action* con los datos necesarios para generar un nuevo *state* de la aplicación.
5. Finalmente, las partes que hayan cambiado de los componentes que estén conectados al *store* de Redux se actualizarán automáticamente.

La estructura de nuestra aplicación se divide principalmente en las siguientes carpetas:

- ***actions***: contiene los archivos Javascript que definen los *actions* y *middleware* de Redux.
- ***api***: contiene los archivos Javascript en los que se realizan las llamadas al API.
- ***assets***: contiene las imágenes y los estilos de la aplicación.
- ***components***: contiene los archivos con los componentes de React reutilizables para toda la aplicación. Corresponde al tipo de componentes de presentación explicados en el Capítulo 5 Tecnologías.
- ***containers***: aquí se encuentran los contenedores de Redux. Estos contenedores conectan componentes (estos componentes serían de tipo *container component*) de React con el *store* de Redux y pasan a través de *props* la información que necesitamos.
- ***layouts***: se definen los diferentes *layouts* de la aplicación.
- ***reducers***: aquí encontramos los archivos con los *reducers* de Redux
- ***routes***: archivos donde se define el enrutamiento de la aplicación.
- ***selectors***: los archivos de esta carpeta sirven para mapear diferentes listas de objetos con el fin de obtener una lista de clave-valor para utilizar en *dropdowns.*
- ***store***: en esta carpeta se encuentran archivos de configuración del *store* de Redux, así como el estado inicial que usaremos en la aplicación (en nuestro caso estará vacío).
- ***utils***: en esta carpeta se almacenan diferentes archivos Javascript que nos sirven de utilidad a la hora de desarrollar. Por ejemplo, funciones para ordenar *arrays.*
- ***views***: se definen las vistas de la aplicación y componentes específicos de las vistas de Usuarios, Parkings y Reservas.

5.2.1 Diagramas de clases

Para mostrar una idea más clara de cómo se relacionan las diferentes clases (componentes) de la IU entre sí, se han realizado diagramas de clases. En estos diagramas aparecen los componentes más significativos. Los componentes reutilizables que se han desarrollado y los proporcionados por librerías como *material-ui* no se han incluido para facilitar la lectura de los diagramas.

5.2.1.1 Layout

Este diagrama de clases representa el *layout* o maquetación de la interfaz de usuario.

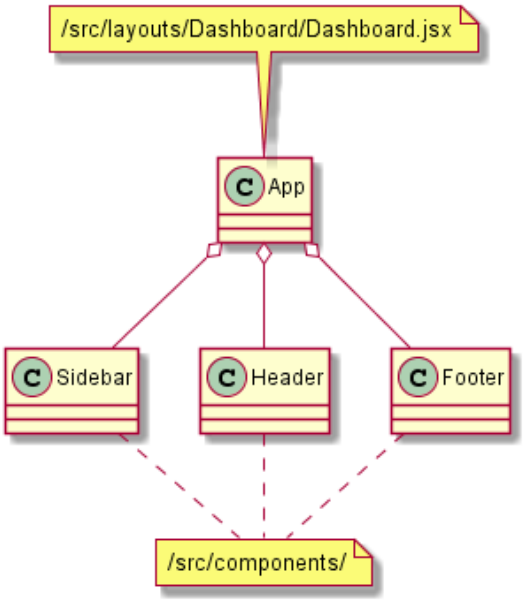

Figura 5-6. Diagrama de clases de layout en IU

5.2.1.2 Parking

El diagrama de la *Figura 5-7* representa las clases relacionadas con los parkings y las plazas de parking.

El componente ParkingView es el componente padre que encapsula otros componentes que forman parte de la pantalla de Parkings. Por ejemplo, podemos ver que tiene como componentes hijos los formularios de parking y plazas de parking (ParkingForm y ParkingSpotForm), un selector de parkings (ParkingSelect) y un listado de plazas de parking (ParkingSpotList)

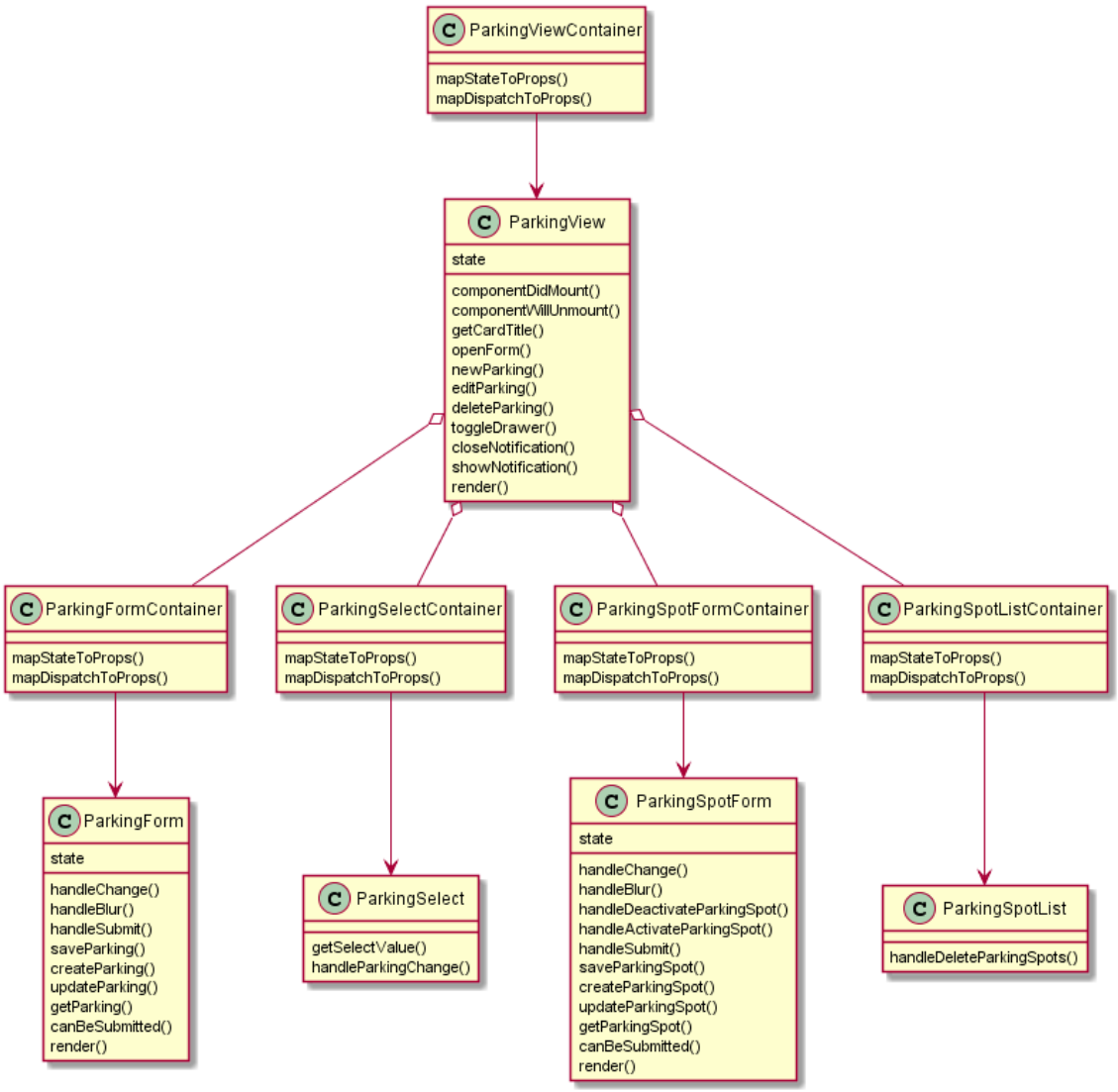

Figura 5-7. Diagrama de clases de parking en IU

5.2.1.3 Usuarios

La *Figura 5-8* muestra el diagrama que representa los componentes relacionados con los usuarios. Así, podemos ver que el componente padre, que es la vista de usuario (UsersView) encapsula otros componentes, como por ejemplo el formulario de usuario (UserForm), y éste a su vez es padre de los componentes de los formularios de pre reservas y ausencias (PreReservationForm y AbsenceForm).

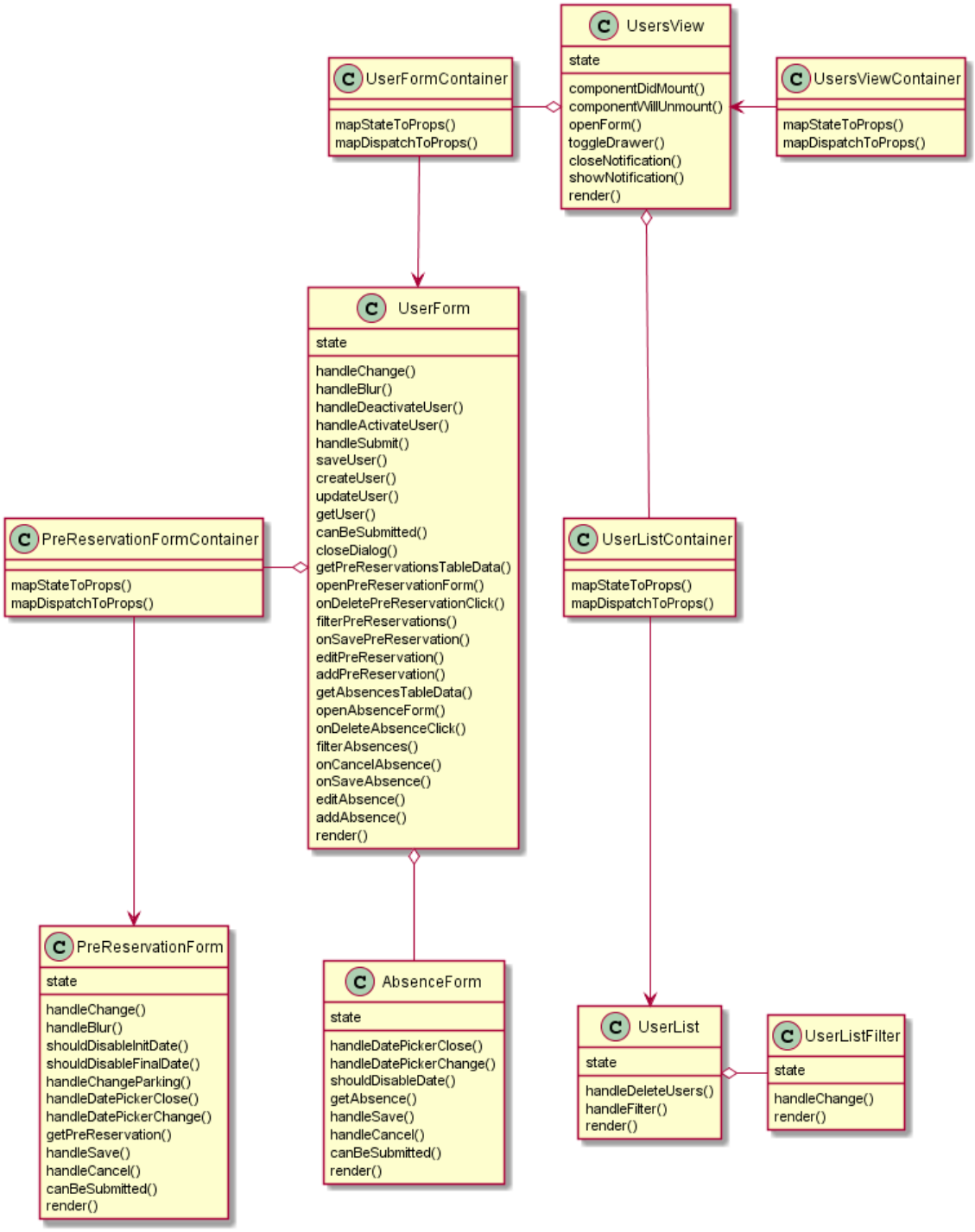

Figura 5-8. Diagrama de clases de usuarios en IU

5.2.1.4 Reservas

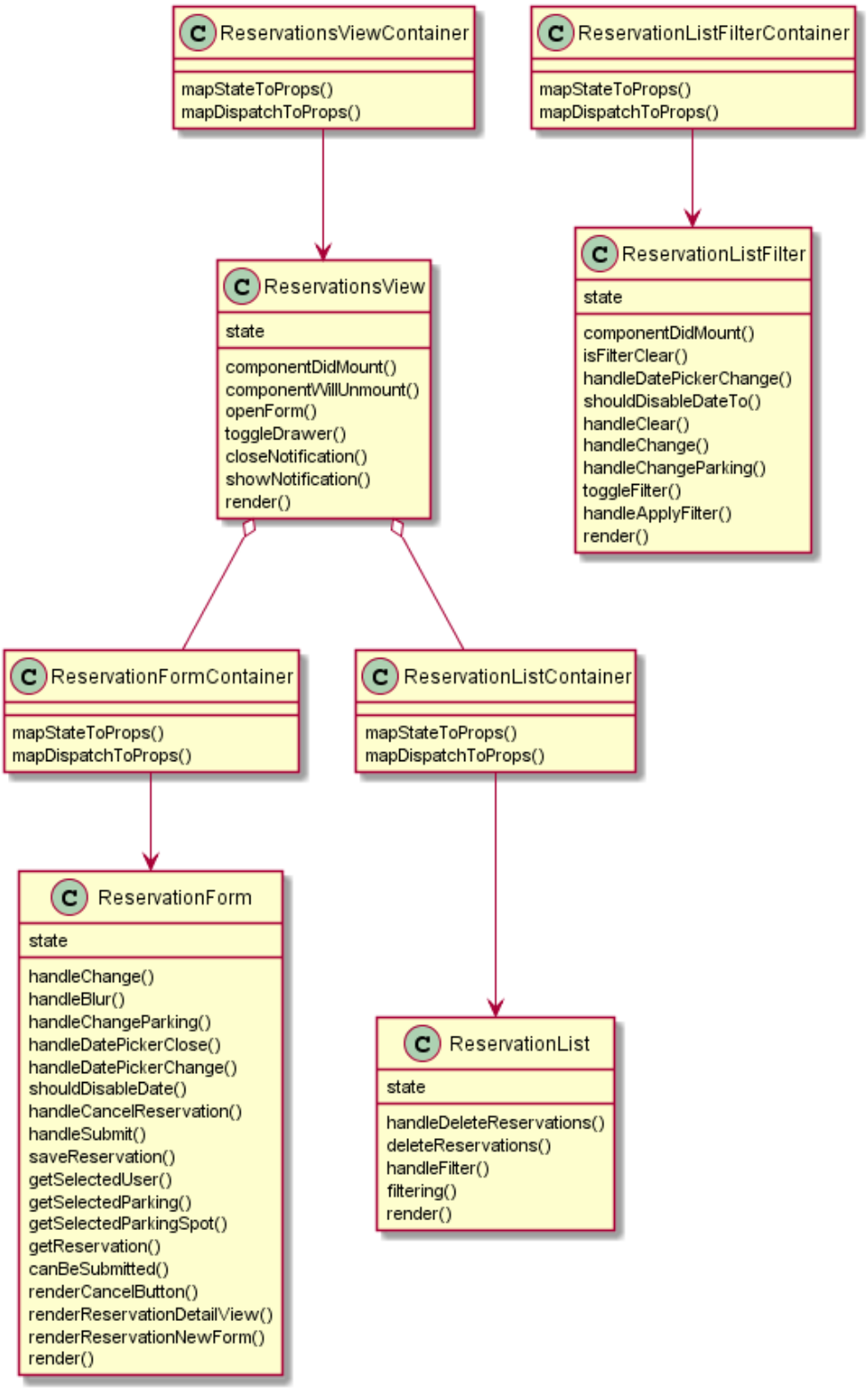

Figura 5-9. Diagrama de clases de reservas en IU

5.2.2 Diagramas de secuencia

Para ilustrar el flujo que se produce en el frontend, se han realizado diagramas de secuencia para detallar cómo interaccionan los diferentes componentes de la interfaz

de usuario desde que se produce un evento hasta que se actualiza el estado de la aplicación.

5.2.2.1 Cargar usuarios de la aplicación

El siguiente diagrama muestra el flujo que sigue la interfaz de usuario desde que el usuario administrador accede a la vista “Usuarios”. Análogamente ocurre en las vistas de “Reservas” y “Parkings”, por lo que se ha decidido no repetir el mismo diagrama. Bastaría con utilizar las clases y archivos correspondientes y la llamada a la API.

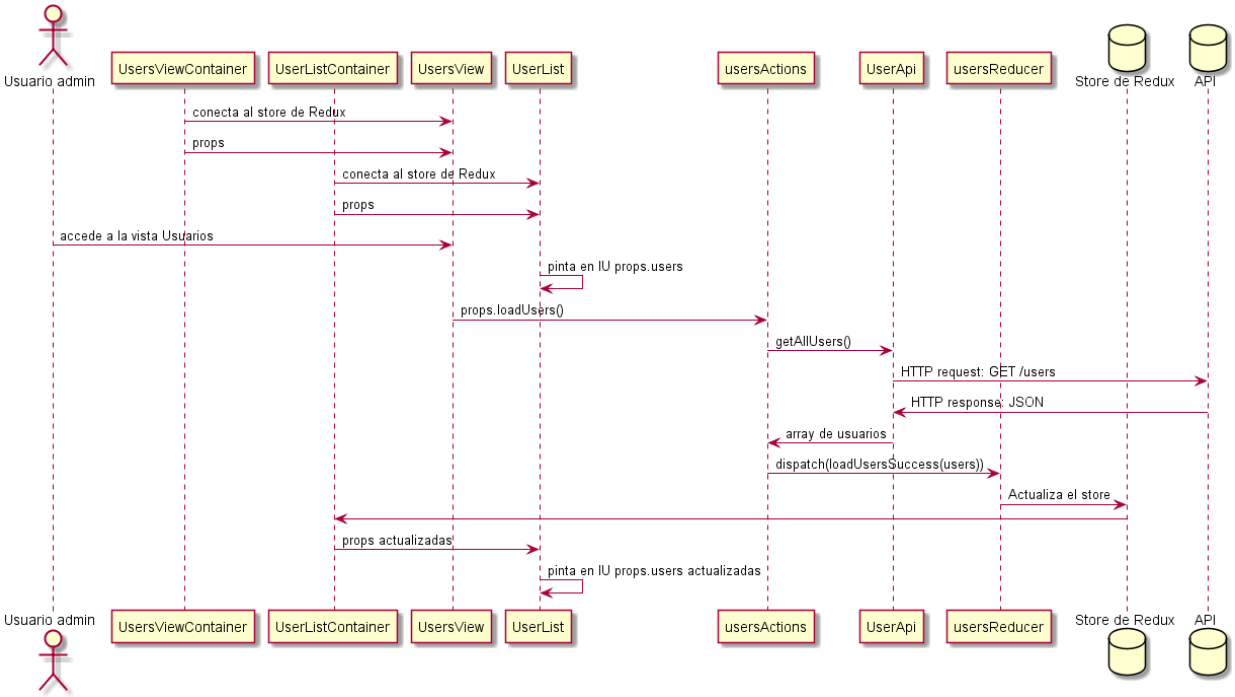

Figura 5-10. Diagrama de secuencia para cargar usuarios en IU

5.2.2.2 Crear usuario de la aplicación

En este diagrama veremos la secuencia de cómo se crea un nuevo usuario en la aplicación. De forma similar sucede al crear reservas o plazas de parking y modificar nuestras entidades, por lo que con este diagrama se pretende mostrar otro ejemplo del flujo que sigue la aplicación en la interfaz de usuario.

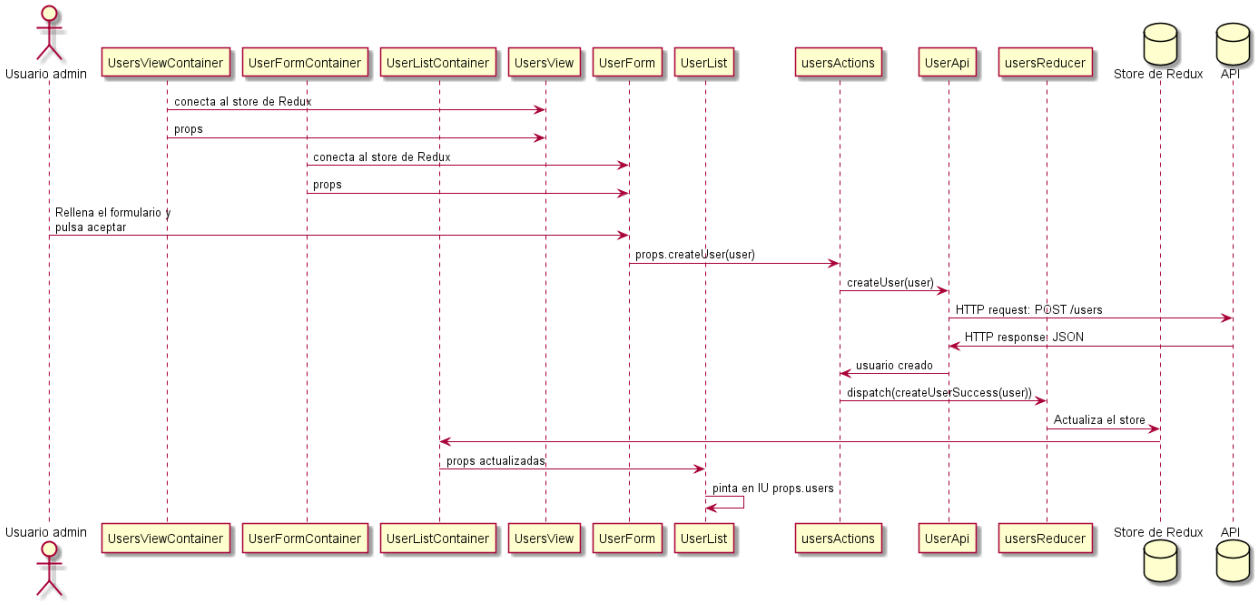

Figura 5-11. Diagrama de secuencia para crear un usuario en IU

5.3 Estructura en Backend

Nuestra aplicación es una aplicación API REST (*Representational State Transfer*).

REST es una interfaz que usa el protocolo HTTP para obtener datos o generar operaciones sobre esos datos en cualquier formato, en nuestro caso JSON.

Las operaciones más importantes relacionadas con los datos son cuatro: POST (crear), GET (leer), PUT (editar) y DELETE (eliminar).

Figura 5-12. Diagrama API REST [31]

El backend de la aplicación de administración se encuentra en el proyecto *parking-admin*. Este proyecto hace uso de *parking-core* para utilizar las entidades y funcionalidad común en todo el sistema.

En *parking-admin* está la lógica de la aplicación y se proporciona una API a *parking-admin-frontend,* donde se ha desarrollado la interfaz de usuario.

La siguiente imagen muestra de forma simplificada el flujo que sigue una petición desde que el usuario la realiza hasta que obtiene una respuesta.

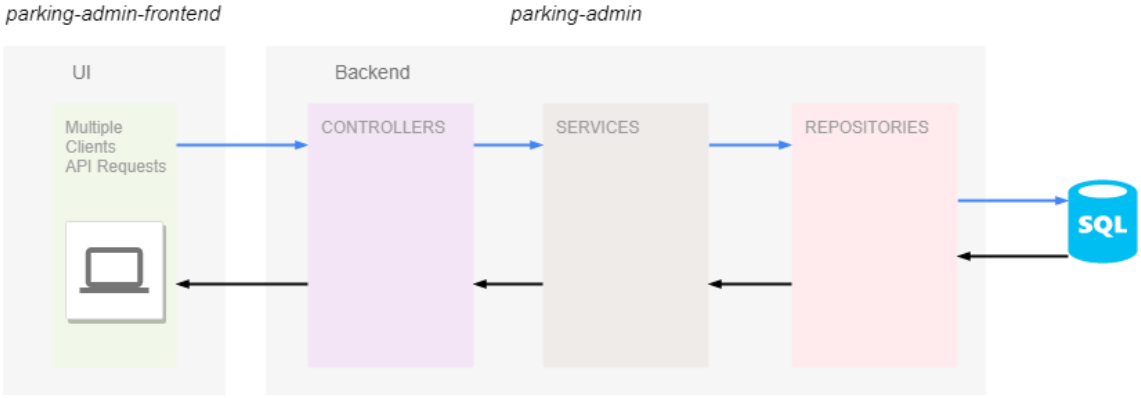

Figura 5-13. Diagrama de estructura en Backend

Al realizar una llamada a la API desde el frontend, la petición es recogida por los *Controller*. Estos se encuentran en el paquete *web.rest* y se encargan de proporcionar al frontend la información necesaria y realizar las funciones CRUD [32]. Son los siguientes:

- *ParkingRestController.* Se encarga de proporcionar la API de los parkings.
- *ParkingSpotRestController.* Proporciona la API para acceder información sobre las plazas de parking.
- *PreReservationRestController.* Proporciona la API sobre las pre-reservas.
- *ReservationRestController.* Se encarga de proporcionar al frontend la interfaz para poder acceder a la información de las reservas y poder realizar las funciones CRUD.
- *UserAppRestController.* Se encarga de proporcionar al frontend una interfaz para acceder a la información de los usuarios de la aplicación y poder realizar las funciones CRUD.

Después de haber realizado la función requerida por el usuario, el *Controller* devuelve la información en el cuerpo de la petición HTTP en formato JSON.

Esta información es mapeada por los DTO (*Data Transfer Object*) de nuestra aplicación para proporcionar al frontend sólo la información necesaria. Los DTOs se encuentran en el paquete *domain.dto.* Además, para mapear fácilmente nuestras entidades de la aplicación a los DTOs y viceversa, tenemos una clases que se encargan de realizar esta conversión, a las que denominamos *Factories,* y que se encuentran en el paquete *domain.factories*.

Los *Controller* requieren de los *Services*, que son las clases que se encargan de realizar la lógica del sistema. Los *Services* encuentran en el paquete *service.impl* y son los siguientes:

- *AbsenceServiceAdminImpl.* Se encarga realizar la lógica relacionada con las ausencias.
- *ParkingServiceAdminImpl.* Realiza la funcionalidad relacionada con los parkings.
- *ParkingSpotServiceAdminImpl.* Realiza la lógica sobre las plazas de parking.
- *PreReservationServiceAdminImpl.* Se encarga realizar la lógica relacionada con las pre-reservas.

- *RequestServiceAdminImpl.* Realiza la lógica relacionada con las solicitudes.
- *ReservationServiceAdminImpl.* Realiza la lógica sobre las reservas.
- *UserAppServiceAdminImpl.* Se encarga de realizar la lógica de los usuarios.

Los *Services* implementan interfaces que están en el paquete *service*. Las inyectan en el contexto de la aplicación para poder tener diferentes implementaciones de los servicios si fuera necesario.

Los *Services* hacen uso de los *Repositories*, que son interfaces que tenemos en el paquete *repositories* e implementan el patrón *Repository* [33]. Estos *Repositories* extienden de la clase JpaRepository que nos proporciona Spring Data. Esto permite tener un repositorio genérico, con la mayoría de funciones que necesitamos para obtener los datos y persistirlos en nuestra base de datos.

5.3.1 Diagramas de clases

5.3.1.1 Entidades

Las entidades utilizadas en la aplicación se encuentran en el paquete *parking-core* debido a que son utilizadas tanto por la aplicación móvil como por la aplicación web de administración, que es la que se desarrolla en este TFG.

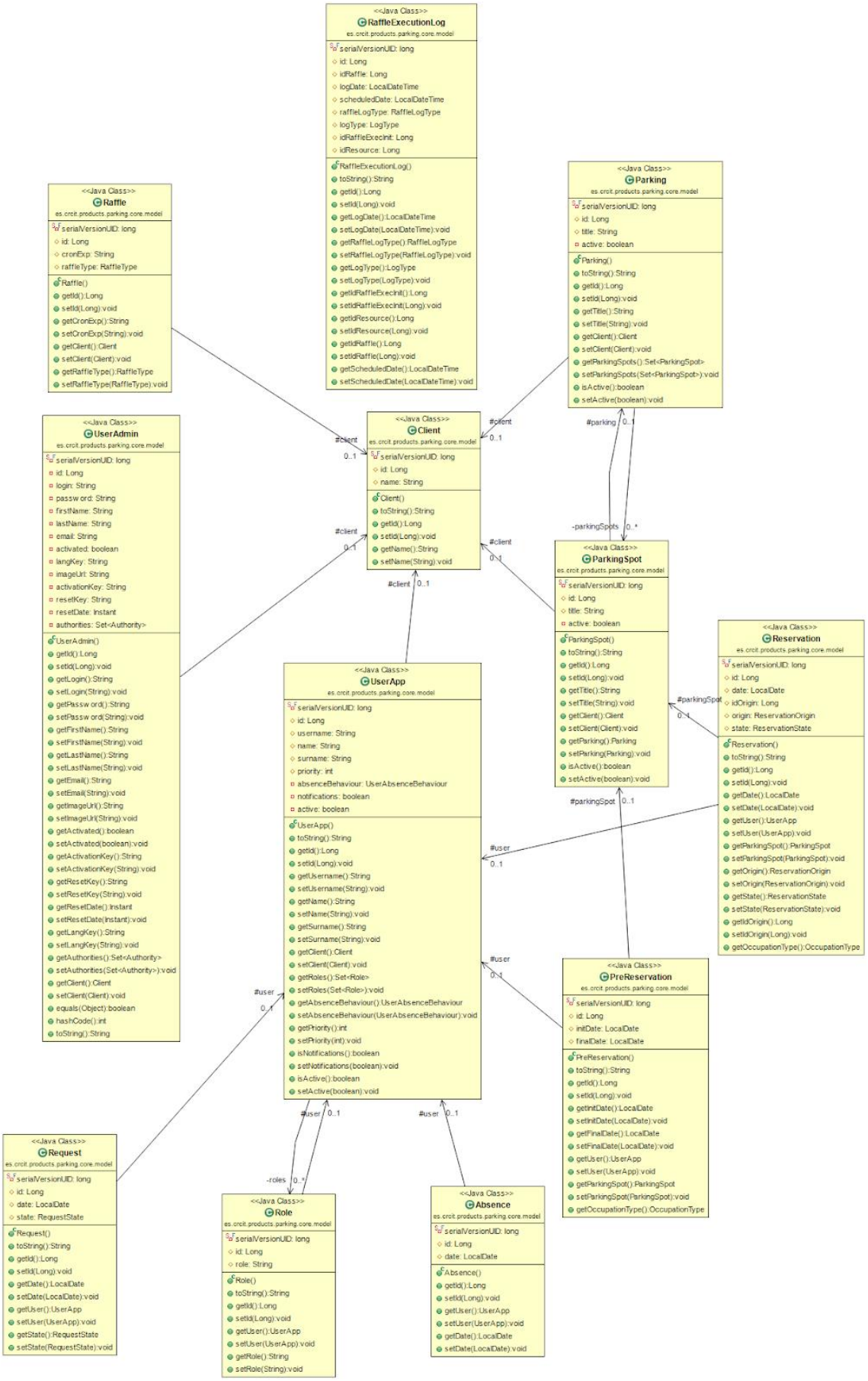

Figura 5-14. Diagrama de clases de las entidades en backend

5.3.1.2 Clases de gestión de parkings

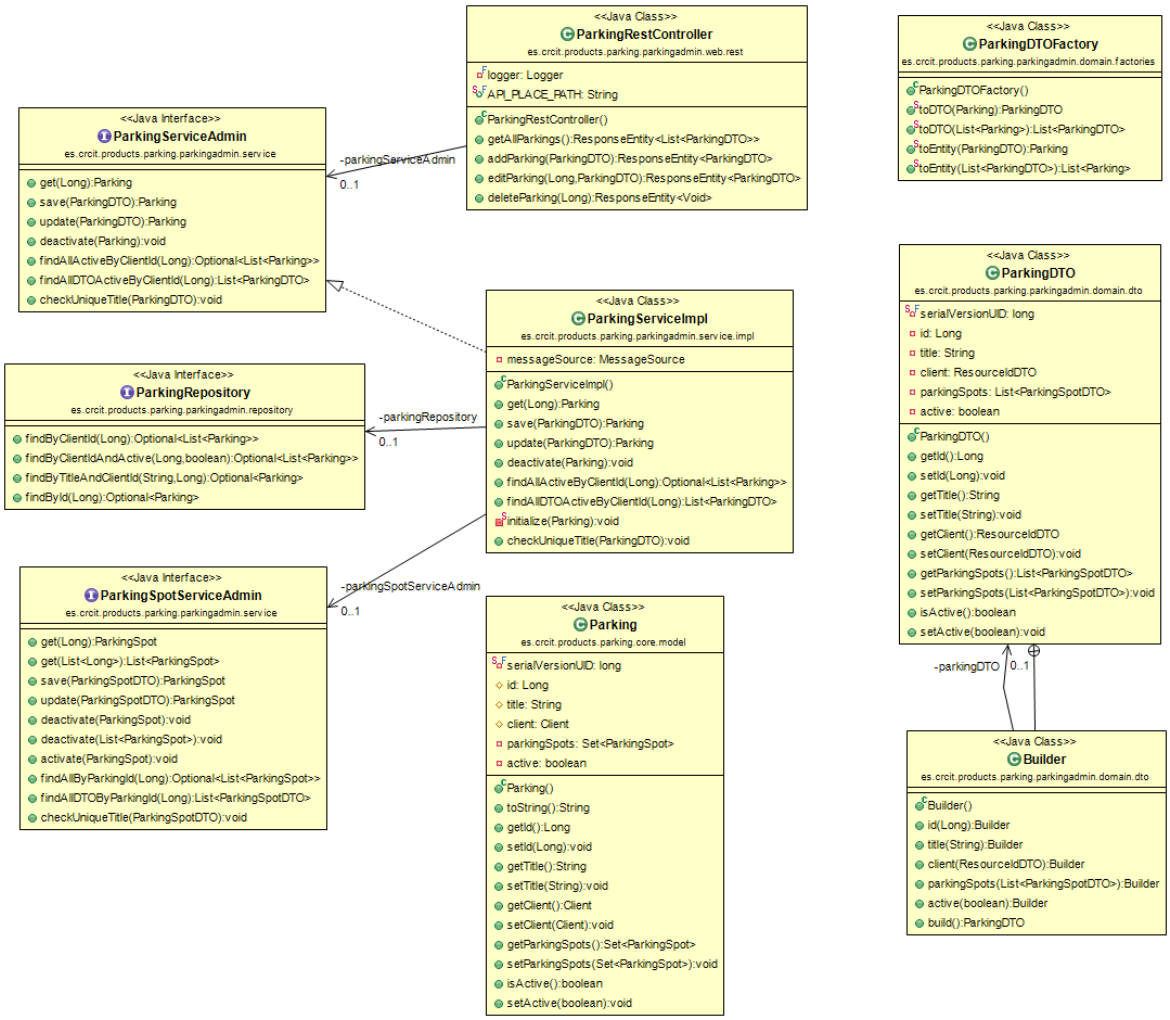

Figura 5-15. Diagrama de clases de gestión de parkings en backend

5.3.1.3 Clases de gestión de plazas de parking

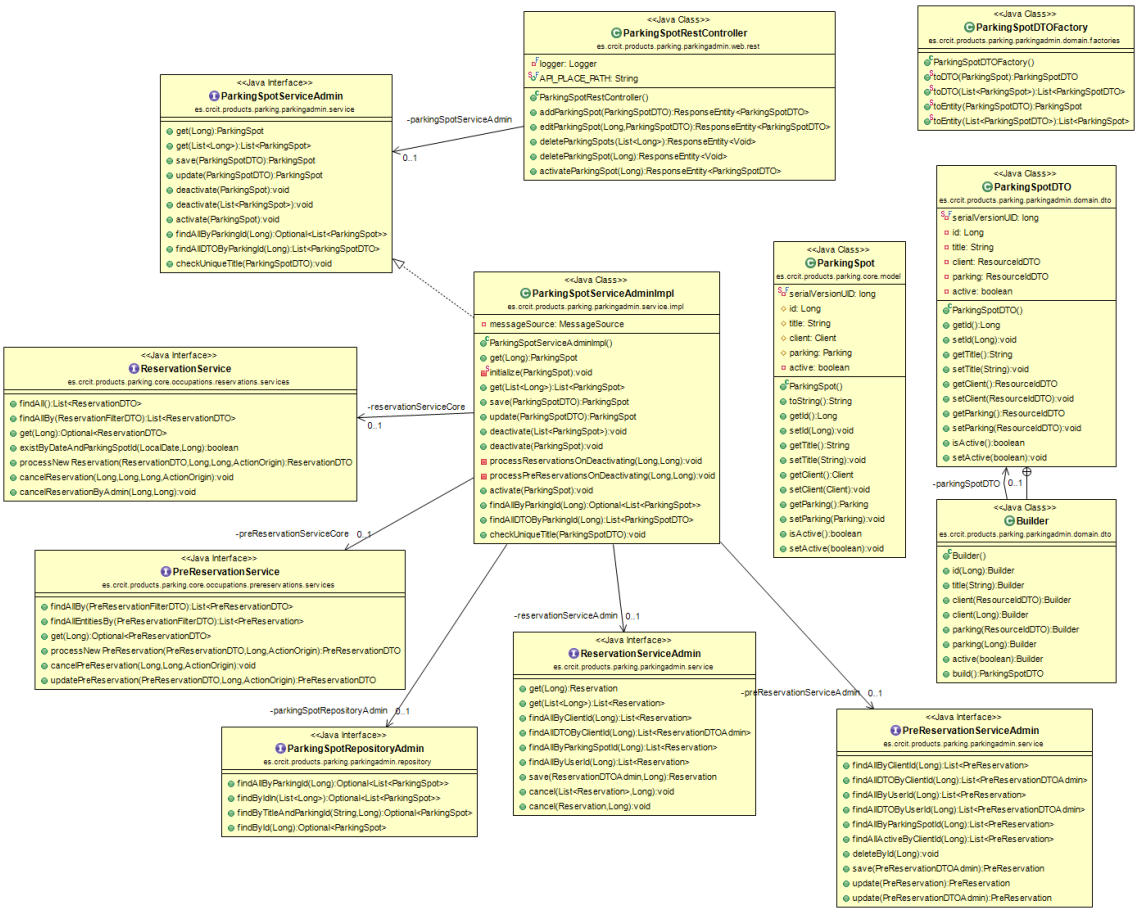

Figura 5-16. Diagrama de clases de gestión de plazas de parking en backend

5.3.1.4 Clases de gestión de usuarios

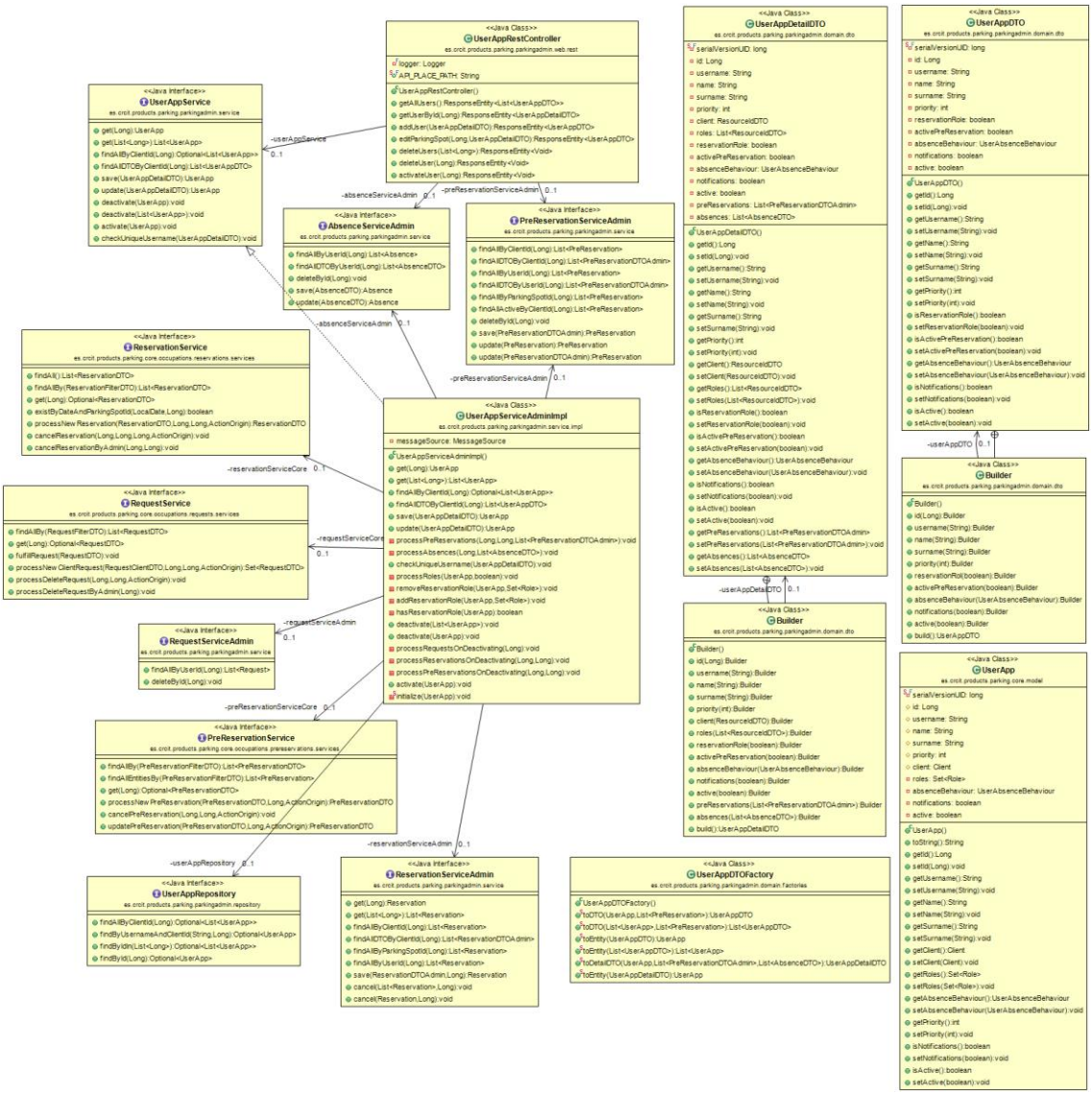

Figura 5-17. Diagrama de clases de gestión usuarios en backend

5.3.1.5 Clases de gestión de pre-reservas

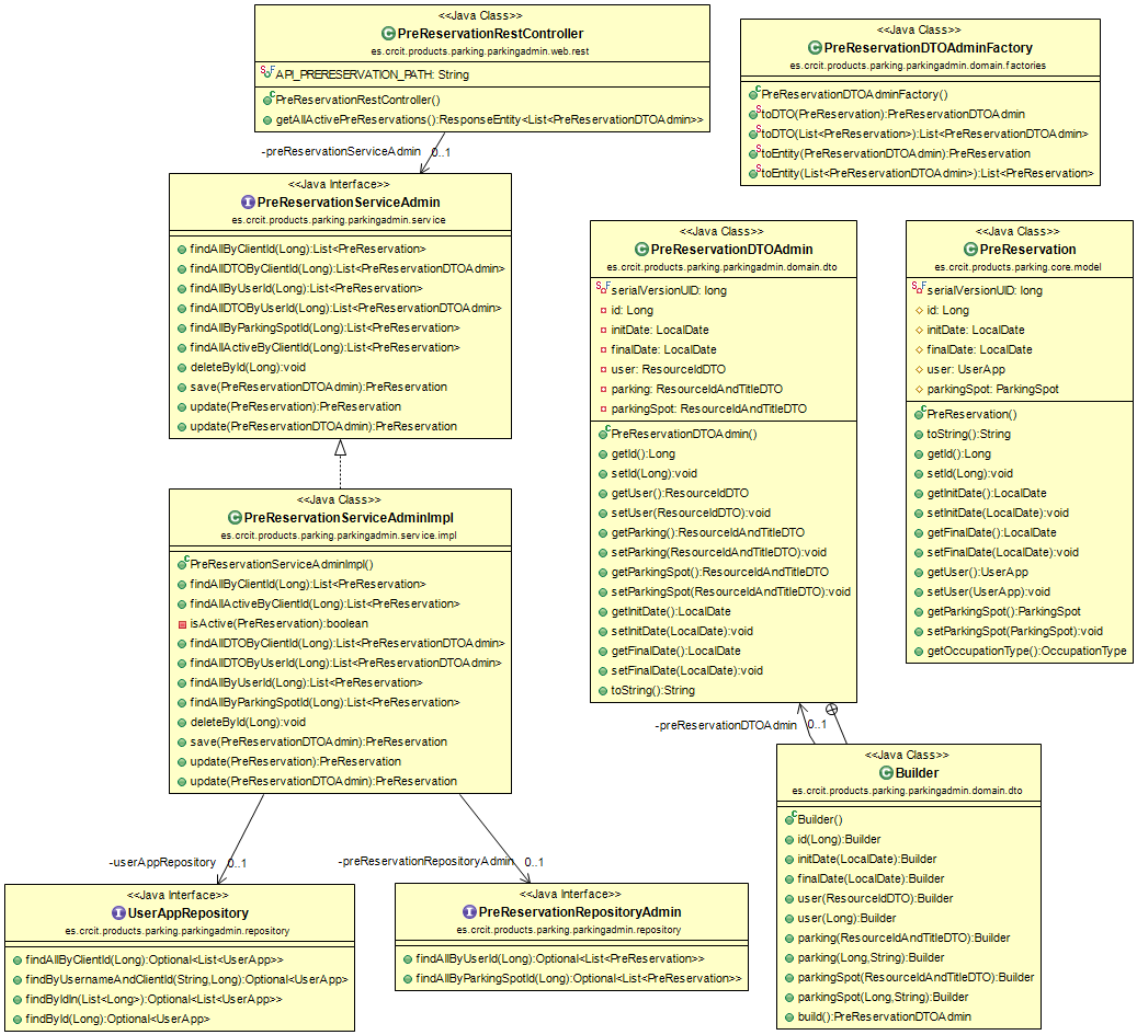

Figura 5-18. Diagrama de clases de gestión de pre-reservas en backend

5.3.1.6 Clases de gestión de ausencias

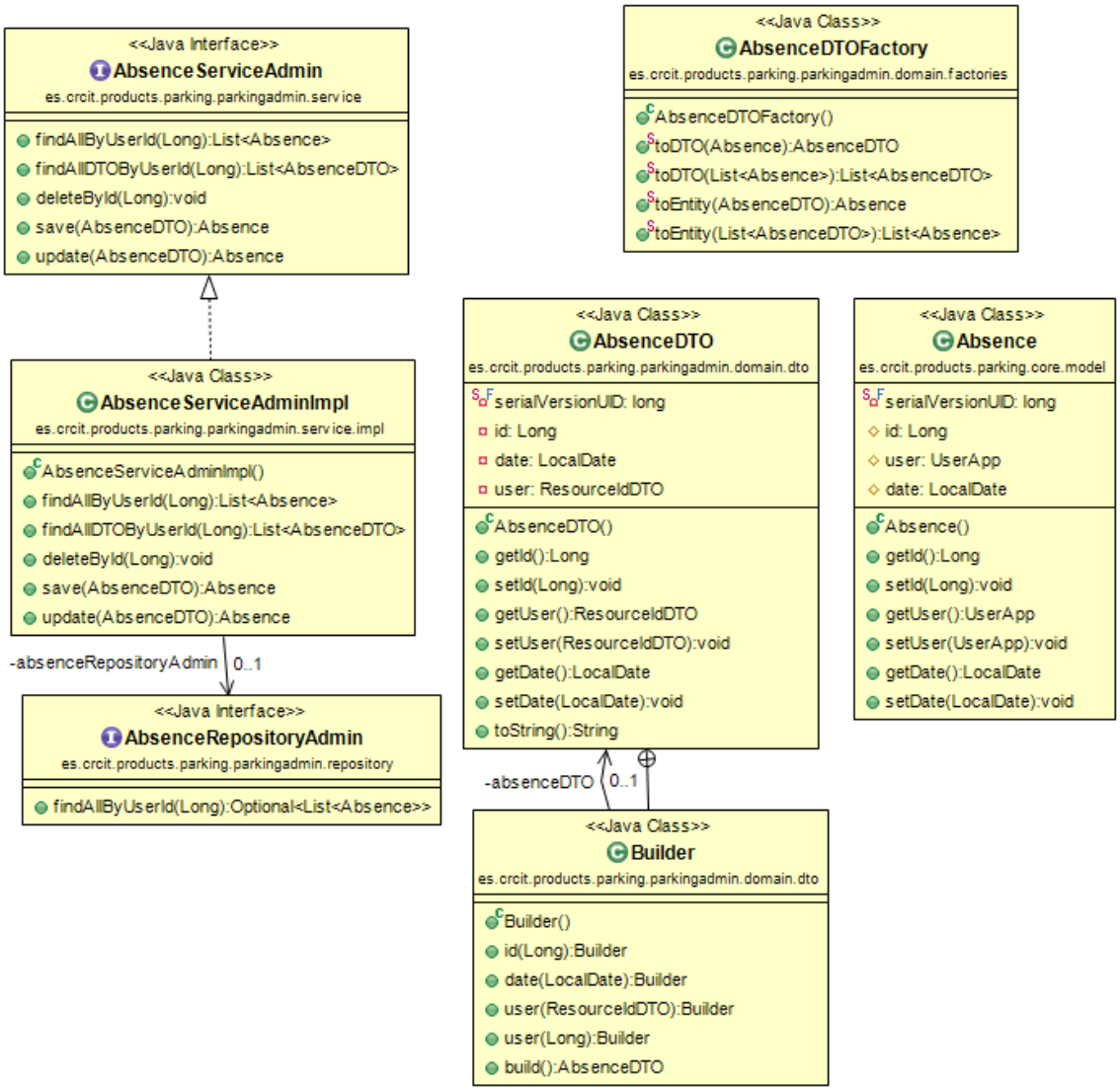

Figura 5-19. Diagrama de clases de gestión de ausencias en backend

5.3.1.7 Clases de gestión de solicitudes

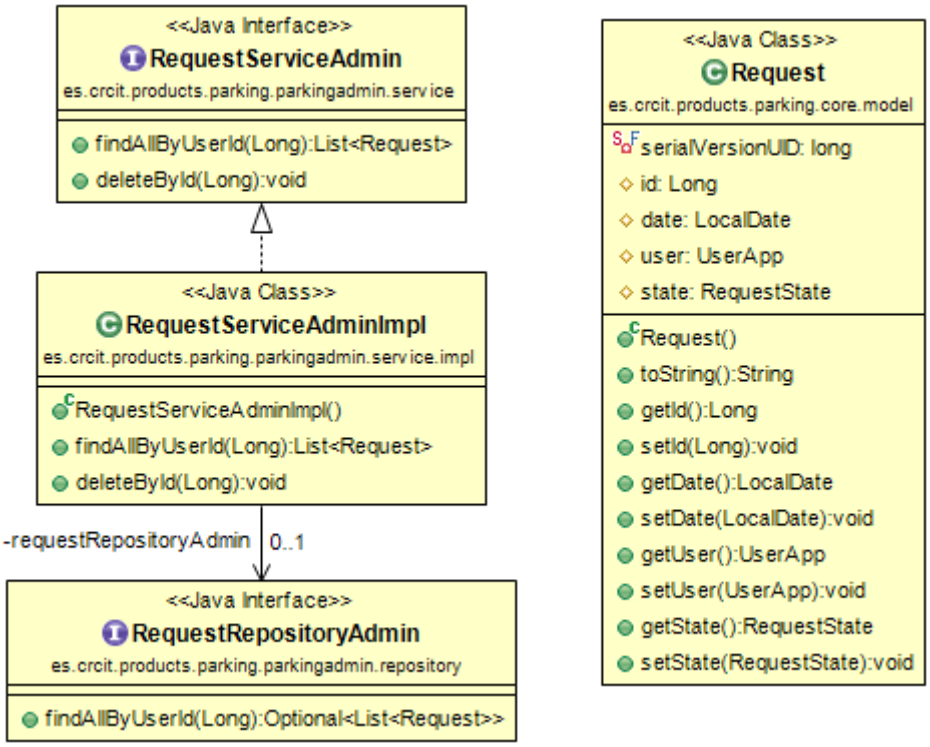

Figura 5-20. Diagrama de clases de gestión de solicitudes en backend

5.3.1.8 Clases de gestión de reservas

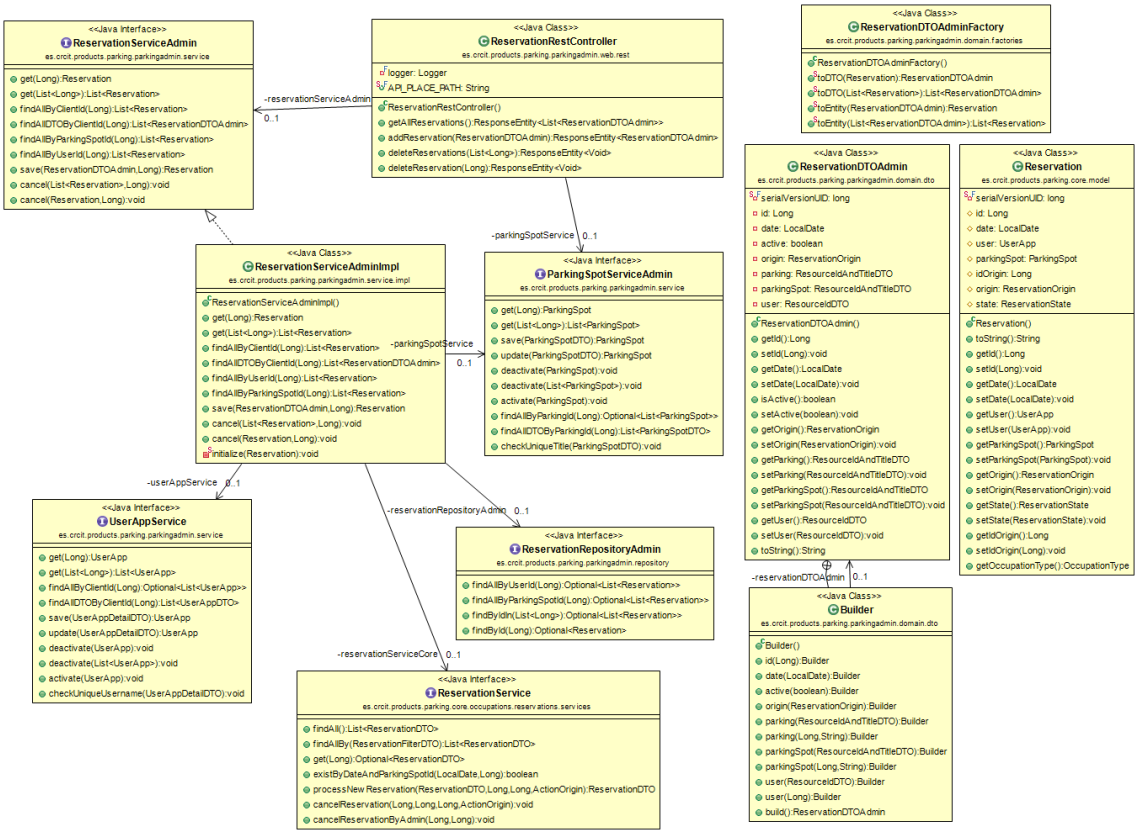

Figura 5-21. Diagrama de clases de gestión de reservas en backend

5.3.2 Diagramas de secuencia

5.3.2.1 Obtención de un listado

En este diagrama se describe cómo se realiza la obtención de un listado de parkings. El diagrama de secuencia es aplicable a la secuencia de obtención del listado de cualquiera de las entidades de la aplicación. Para no repetir el diagrama para todas y cada una de las llamadas a la API, solamente se muestra el ejemplo para la obtención del listado de parkings.

El siguiente diagrama (ver *Figura 5-22*) muestra la secuencia cuando el usuario hace una petición a la API de Parkings de tipo GET */parkings*, concretamente al método getAllParkings() de la clase ParkingRestController.

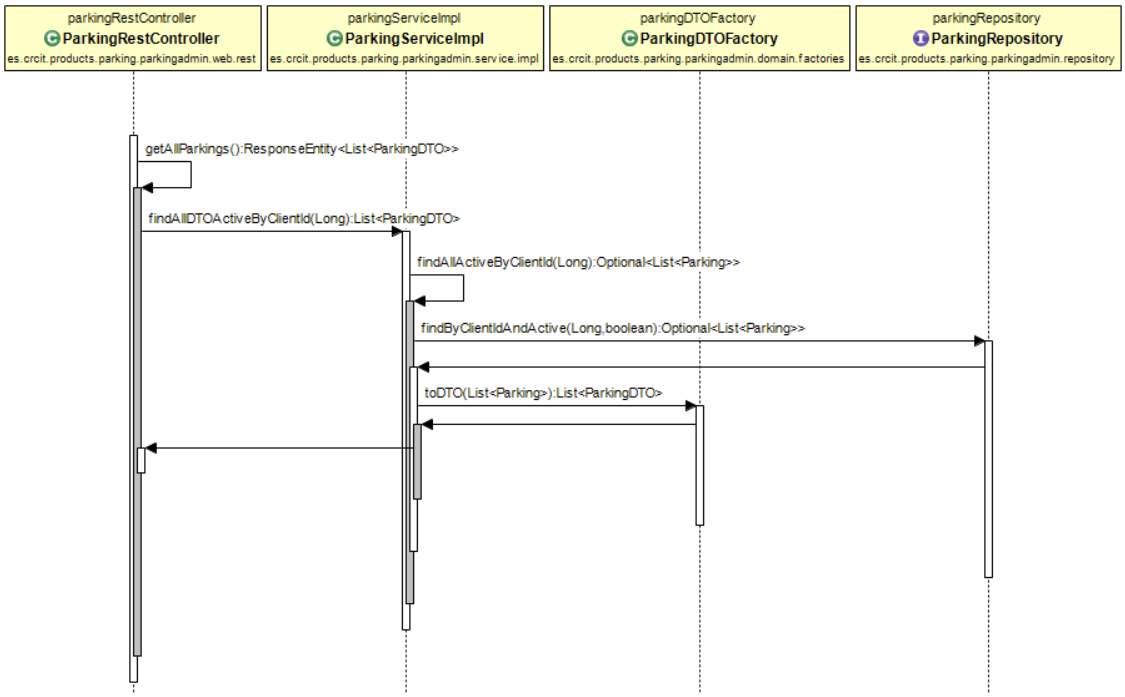

Figura 5-22. Diagrama de secuencia para la obtención del listado de parkings

5.3.2.2 Crear un recurso

Este diagrama describe la creación de un usuario. La forma en la que se crea un recurso en la aplicación es similar para todas las entidades, por eso nuevamente sólo se muestra el ejemplo de creación de un usuario.

En el diagrama se muestra la secuencia para cuando el usuario hace una petición a la API de Usuarios de tipo POST */users*, concretamente al método addUser(UserAppDetailDTO userDetail) de la clase UserAppRestController:

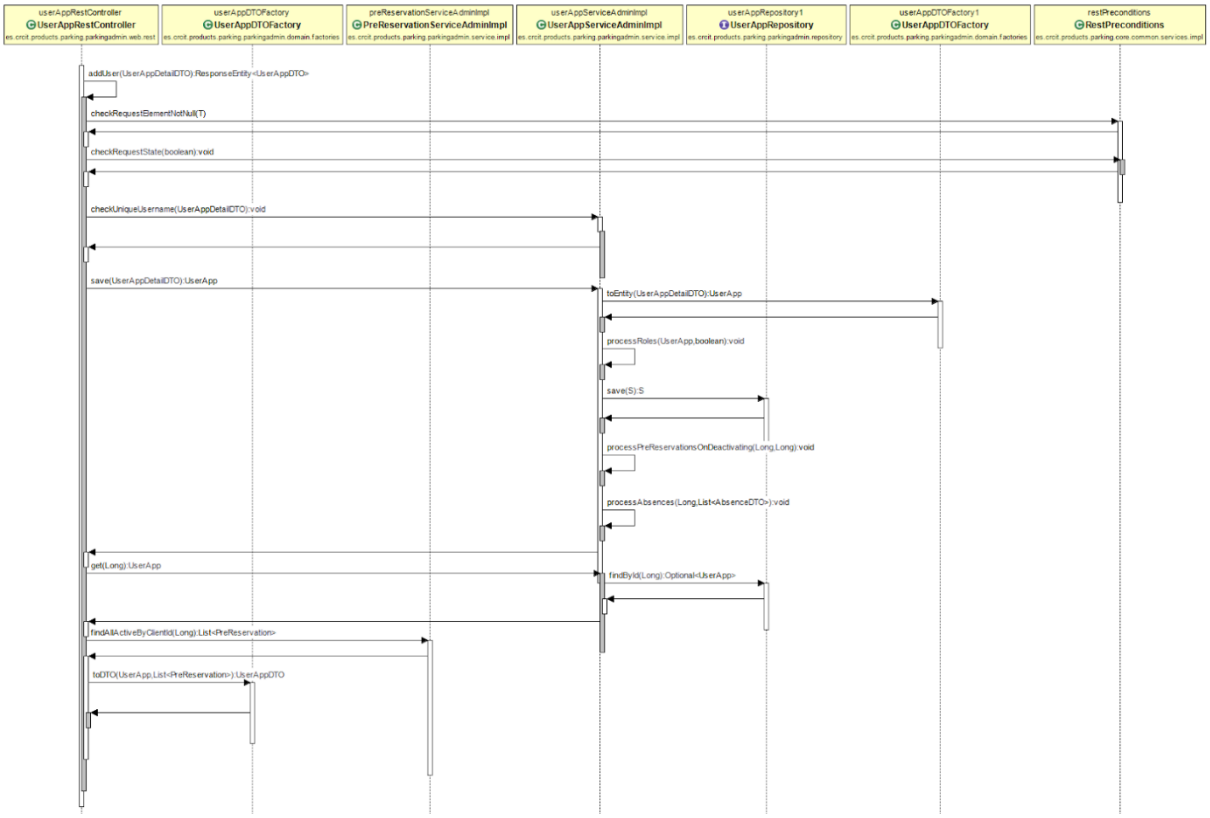

Figura 5-23. Diagrama de secuencia para crear un usuario

5.3.2.3 Modificar un recurso

La secuencia para modificar un recurso es muy similar a la de crear un recurso. En este caso se muestra el ejemplo de cancelar una reserva.

El usuario hace una petición a la API de Reservas de tipo PUT */delete/{id_de_la_reserva}*, concretamente al método deleteReservation(Long id) de la clase ReservationRestController:

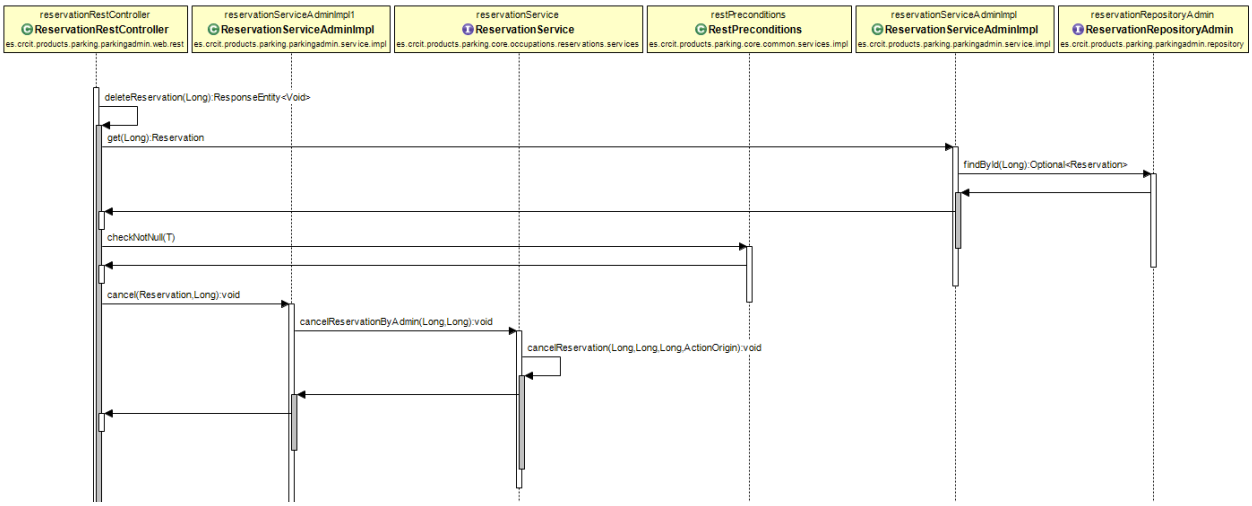

Figura 5-24. Diagrama de secuencia para cancelar una reserva

5.3.2.4 Eliminar un recurso

Realmente en la aplicación no se elimina ningún recurso, ya que el borrado de todas las entidades es un borrado lógico.

Aun así, la forma que tendría la llamada a la API para, por ejemplo, eliminar un parking, sería DELETE */parkings/{id}.*

5.4 Base de datos

Utilizamos Liquibase para generar el esquema de base de datos a partir de las entidades de nuestra aplicación. La *Figura 5-25* muestra el diagrama Entidad-Relación de la base de datos, generado con MySQL Workbench:

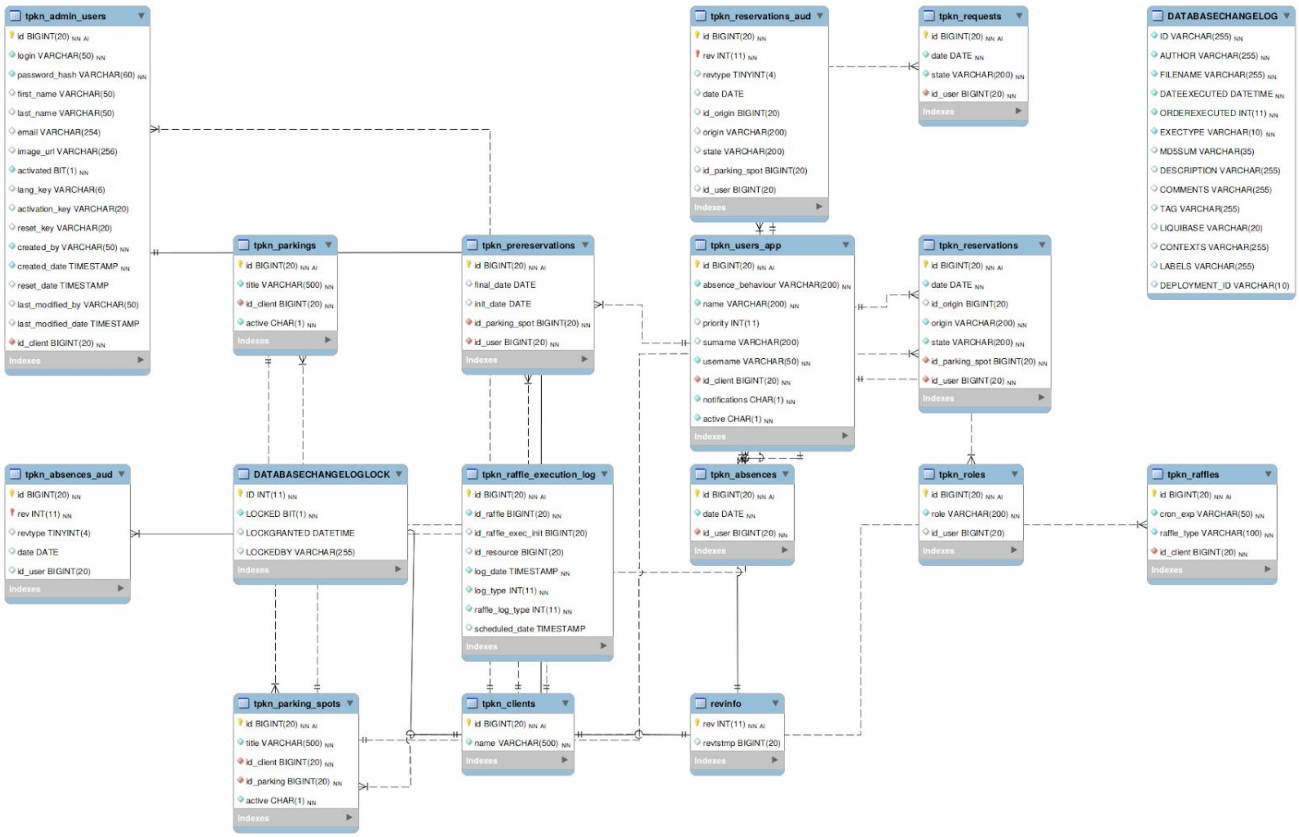

Figura 5-25. Diagrama ER (Entidad-Relación)

Todas las tablas contienen el campo ID como clave primaria para identificar cada registro. A continuación, se describen los datos que almacenan las tablas que componen la base de datos:

- **Tabla *tpkn_absences***: Ausencias de los usuarios.
- **Tabla *tpkn_admin_users***: Usuarios administradores.
- **Tabla *tpkn_clients***: Clientes.
- **Tabla *tpkn_parking_spots***: Plazas de parking.
- **Tabla *tpkn_parkings***: Ausencias de los usuarios.
- **Tabla *tpkn_preresevations***: Pre reservas de los usuarios.

- **Tabla *tpkn_raffle_execution_log***: En esta tabla se guardan, para cada ejecución que se produzca de un sorteo, un *log* al inicio del sorteo, y otro *log* al final de la ejecución del sorteo.
- **Tabla *tpkn_raffles***: Es la tabla donde se almacenan diferentes sorteos del sistema.
- **Tabla *tpkn_requests***: Solicitudes realizadas por los usuarios.
- **Tabla *tpkn_reservations***: Reservas que tienen los usuarios.
- **Tabla *tpkn_roles***: Almacena los roles de los usuarios de la aplicación. Actualmente sólo existe el rol de “Reserva”. Éste permite confiere a un usuario de la aplicación el permiso de realizar directamente una reserva sin hacer antes una solicitud. Esta tabla se ha creado para añadir diferentes roles en futuras versiones.
- **Tabla *tpkn_users_app***: Usuarios de la aplicación.

Capítulo 6 - Interfaz de usuario y funcionalidad

En este capítulo se describe la funcionalidad de la aplicación web de administración, detallando la interfaz de usuario.

Las partes de la interfaz de usuario en las que nos centraremos en este capítulo son los siguientes:

- *Layout*
- Autenticación
- Gestión de parkings y plazas de parking
- Gestión de usuarios
- Gestión de reservas

6.1 Layout

En esta sección se muestra el menú lateral (ver *Figura 6-2*) y cómo está maquetada la aplicación. El menú está presente en todas las pantallas y a través de él podemos acceder a las secciones Parking, Usuarios y Reservas. Además, en la cabecera siempre tenemos disponible el botón de cerrar sesión (ver *Figura 6-1*).

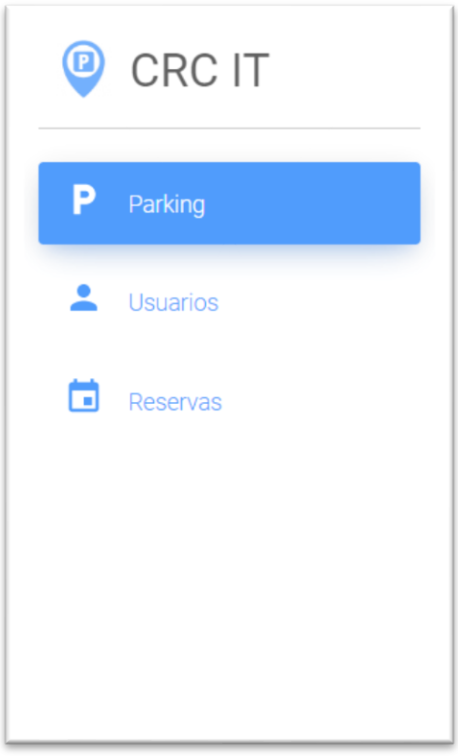

Figura 6-2. Captura de pantalla de IU: Menú lateral

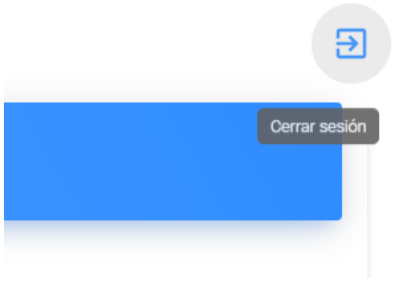

Figura 6-1. Captura de pantalla de IU: Botón para cerrar sesión

6.2 Autenticación

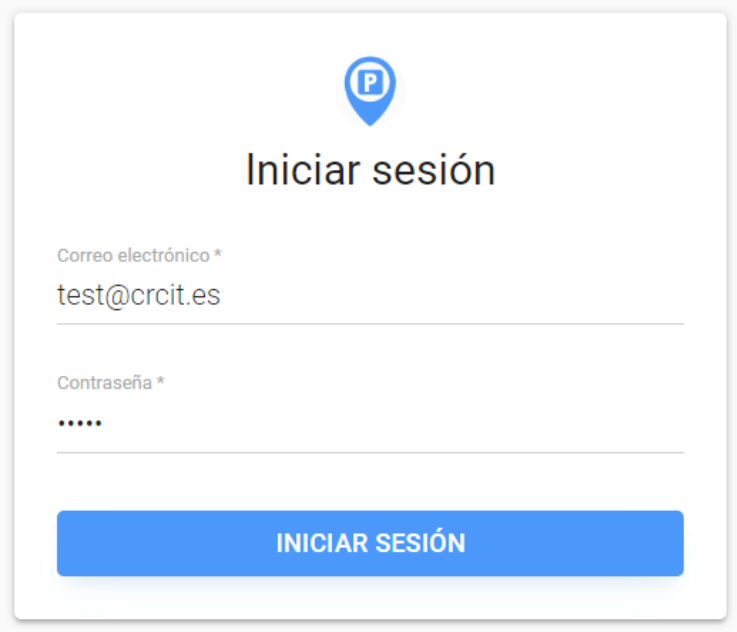

Figura 6-3. Captura de pantalla de IU: Iniciar sesión

Habrá una página de *Login* para iniciar sesión en la aplicación. Tras escribir el nombre de usuario (correo electrónico) y la contraseña y hacer click en el botón "INICIAR SESIÓN", accederemos a la aplicación a la pantalla de *Parkings*.

6.3 Gestión de parkings y plazas de parking

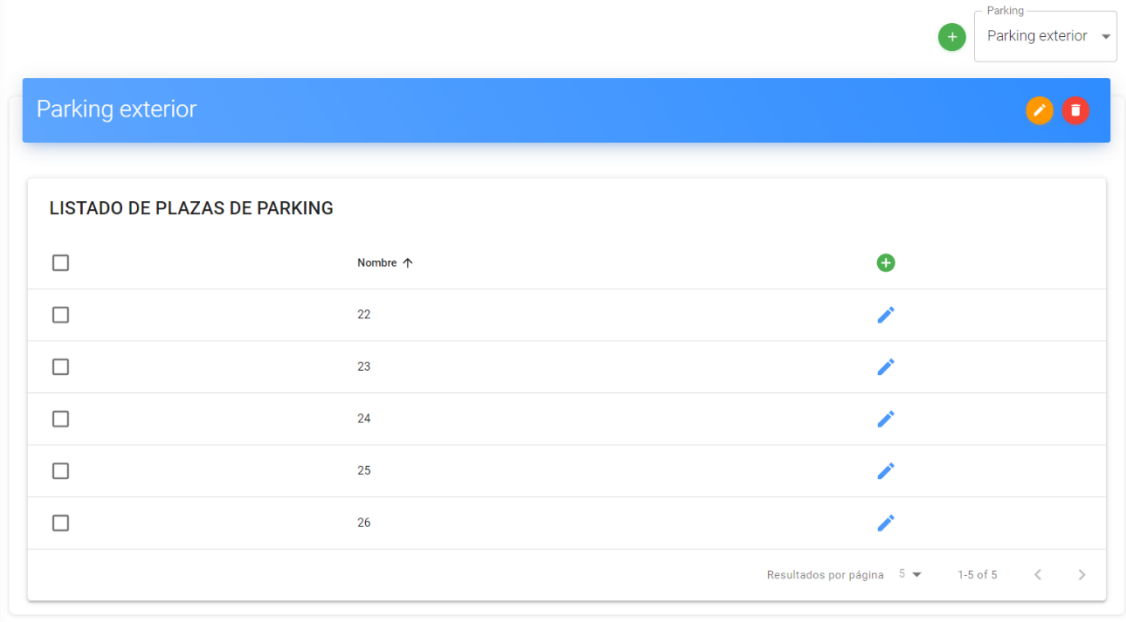

Figura 6-4. Captura de pantalla de IU: Listado de parkings

Los usuarios de administración podrán realizar las siguientes acciones:

- Consultar los parkings existentes y las plazas de parking que se encuentran en cada parking. Tanto los parkings como las plazas de parking solo tienen como atributo un nombre.
- Ordenar las plazas de parking en orden ascendente y descendente.
- Crear, modificar y eliminar parkings.
- Crear, modificar y desactivar plazas de parking.

Los formularios para crear y modificar parkings y plazas de parking son muy similares. Como ejemplo, se muestra el formulario de edición de una plaza de parking:

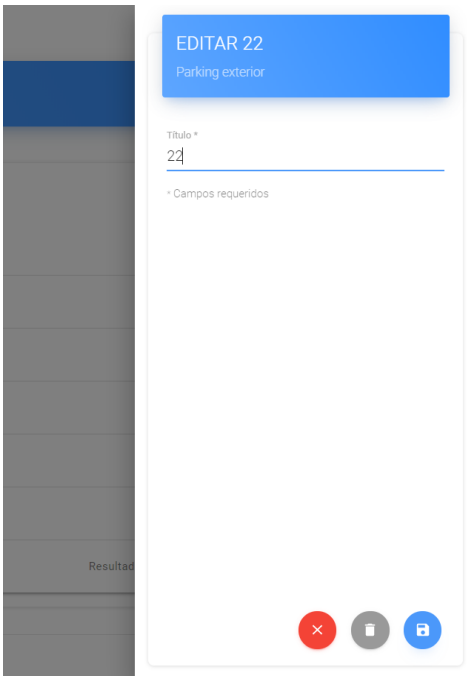

Figura 6-5. Captura de pantalla de IU: Formulario de plazas de parking

Al eliminar un parking, se le muestra al administrador un mensaje de confirmación:

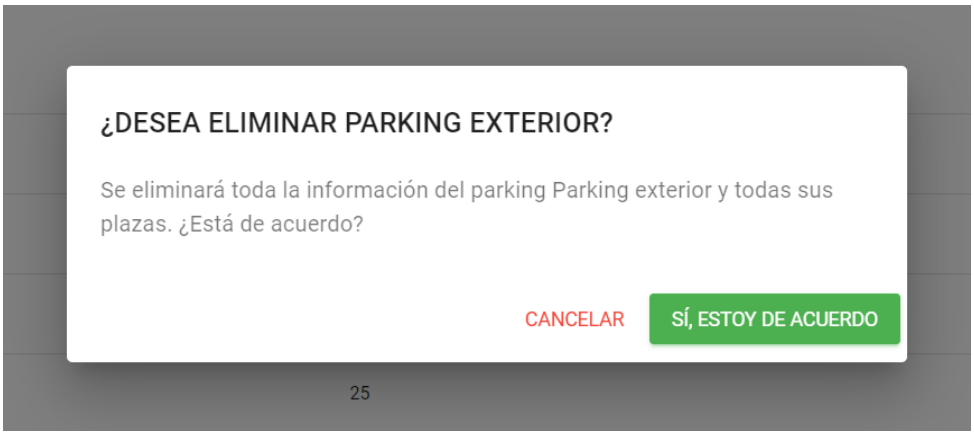

Figura 6-6. Captura de pantalla de IU: Confirmación para eliminar un parking

Además, en la *Figura 6-7* podemos ver cómo se muestran las plazas de parking inactivas y cómo el administrador puede seleccionar varias plazas de parking para darlas de baja masivamente:

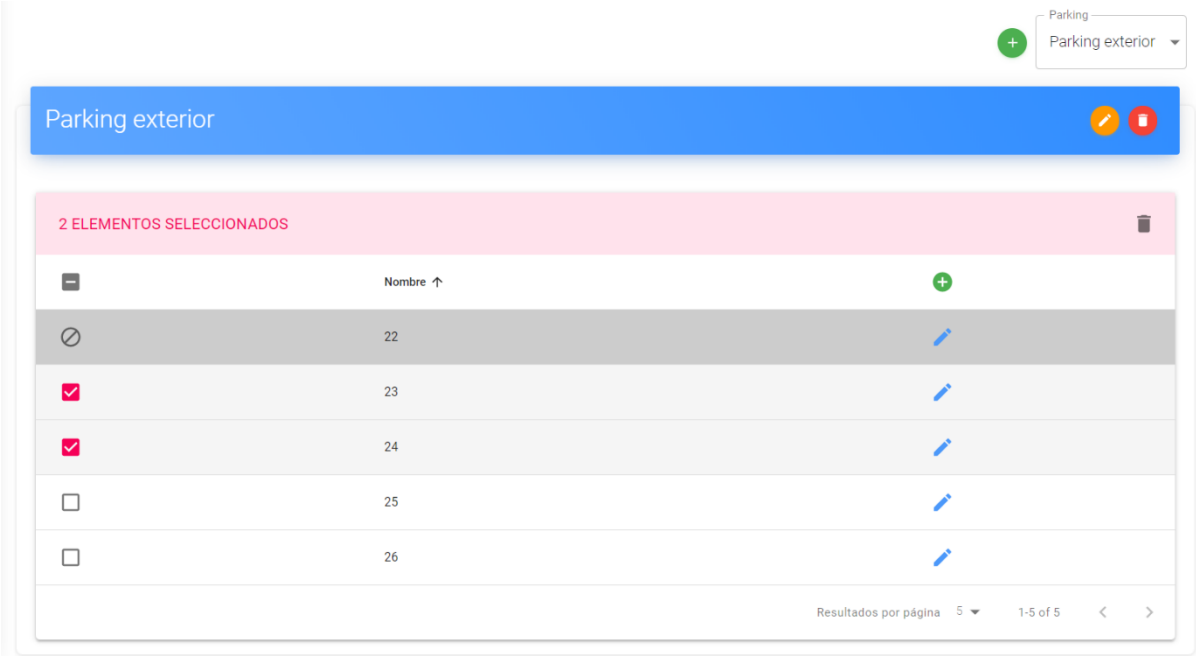

Figura 6-7. Captura de pantalla de IU: Plazas de parking seleccionadas e inactivas

6.4 Gestión de usuarios

Un administrador podrá consultar los usuarios de la aplicación. La información a consultar será: nombre de usuario, nombre, apellidos, si tiene permiso para reservar, si tiene pre-reservas y prioridad. La tabla tiene un filtrado para mostrar los usuarios inactivos y ofrece la posibilidad de ordenar por cada uno de los campos en orden ascendente y descendente.

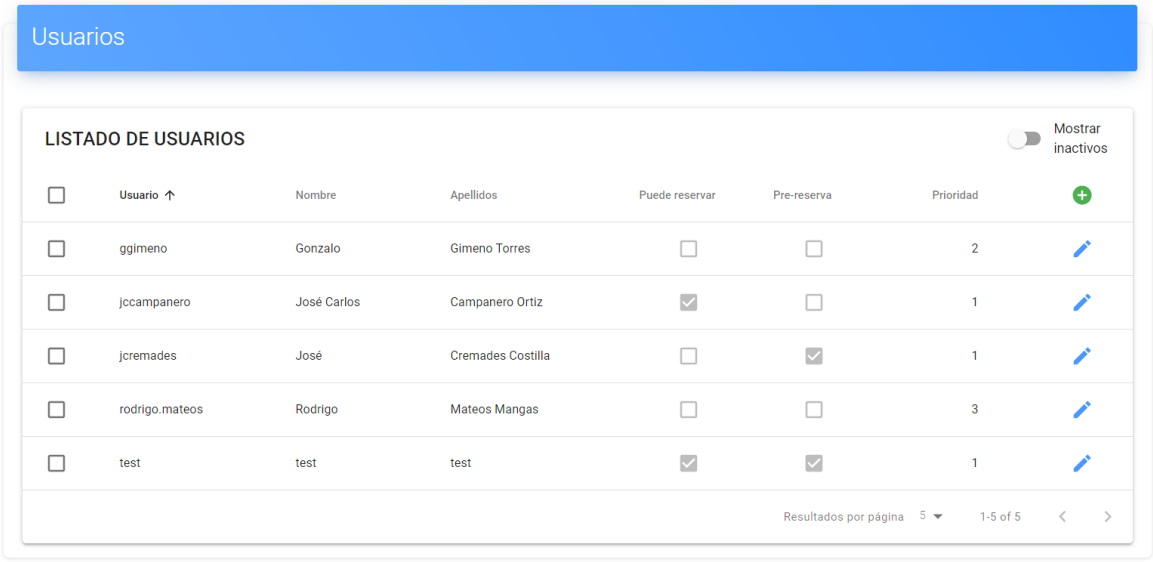

Figura 6-8. Captura de pantalla de IU: Listado de usuarios

La forma en que se visualizan los usuarios inactivos y la forma en la que se seleccionan varios usuarios para darlos de baja masivamente es análogo a la gestión de plazas de parking (ver *Figura 6-7*).

Al seleccionar un usuario, la información a consultar, crear y/o editar en su detalle es:

- *Username* (no editable en caso de que se esté modificando un usuario existente).
- Nombre.
- Apellidos.
- Prioridad.
- Si libera plaza en caso de ausencia.
- Si tiene permiso de reserva.
- Si tiene activadas las notificaciones.
- Lista de pre-reservas del usuario.
- Lista de ausencias del usuario

Las capturas que podemos observar en la *Figura 6-10* y la *Figura 6-9* muestran las diferentes pestañas del formulario para editar un usuario:

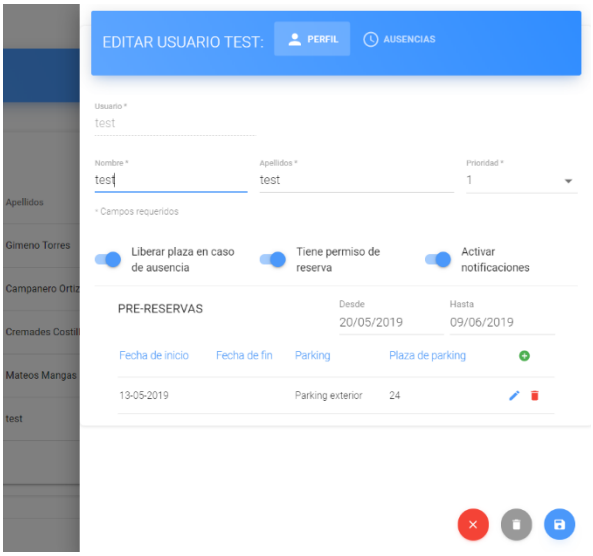

Figura 6-10. Captura de pantalla de IU: Pestaña "Perfil" del formulario de usuario

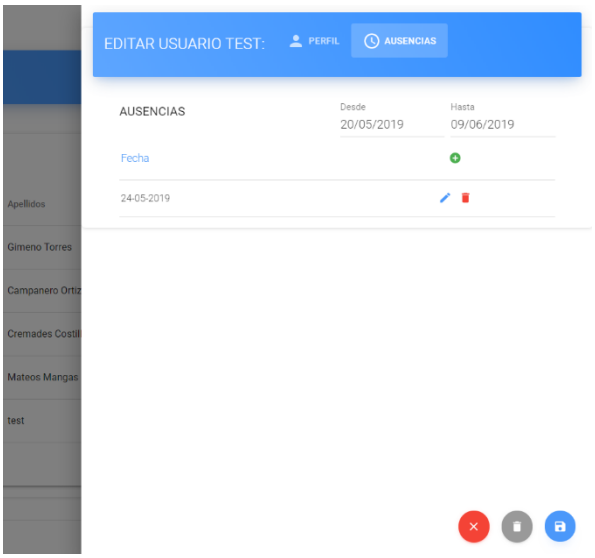

Figura 6-9. Captura de pantalla de IU: Pestaña "Ausencias" del formulario de usuario

En cuanto a la gestión de pre-reservas y ausencias del usuario, tienen la posibilidad de filtrar por un rango de fechas. La siguiente figura muestra los formularios de edición de ausencias y pre-reservas, muy similares a los formularios de alta:

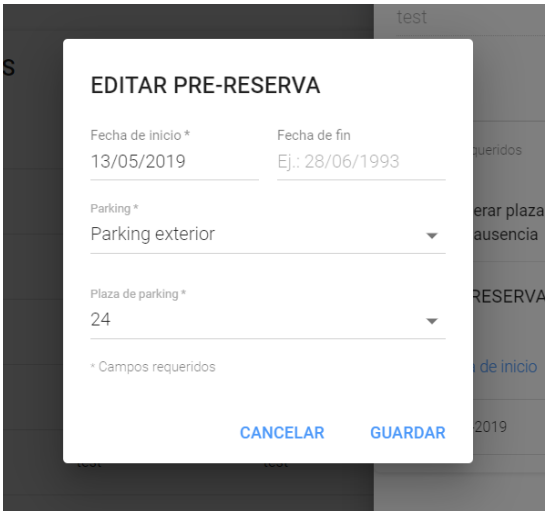

Figura 6-12. Captura de pantalla de IU: Formulario de pre-reserva

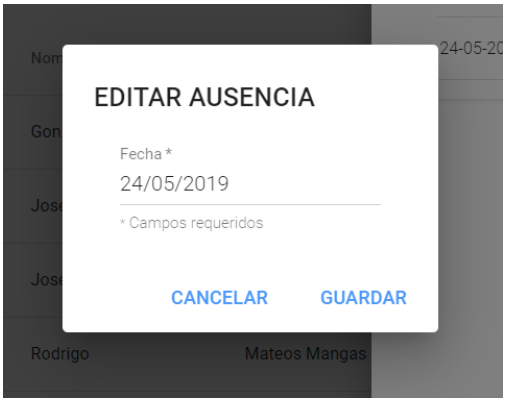

Figura 6-11. Captura de pantalla de IU: Formulario de ausencia

Al seleccionar la opción de dar de baja usuario, ya sea desde el formulario de edición o masivamente desde el listado:

- Se realizará una baja lógica, marcando el usuario como inactivo.
- Las pre-reservas del usuario que tengan una fecha de inicio posterior al día actual serán eliminadas.
- A las pre-reservas del usuario que estén activas, es decir, que tengan una fecha de inicio igual o anterior al día actual, se le establecerá como fecha de fin el día en el que se realiza la baja de usuario.
- Las reservas del usuario posteriores al día actual se marcarán como canceladas.
- Las solicitudes del usuario posteriores al día actual se marcarán como canceladas.

6.5 Gestión de reservas

Un administrador puede consultar las reservas existentes. La información a consultar es fecha, parking, plaza de parking, nombre de usuario y origen de la reserva (puede ser 'Reserva', 'Pre reserva' o 'Solicitud') y ofrece ordenación por cada uno de los campos en orden ascendente y descendente.

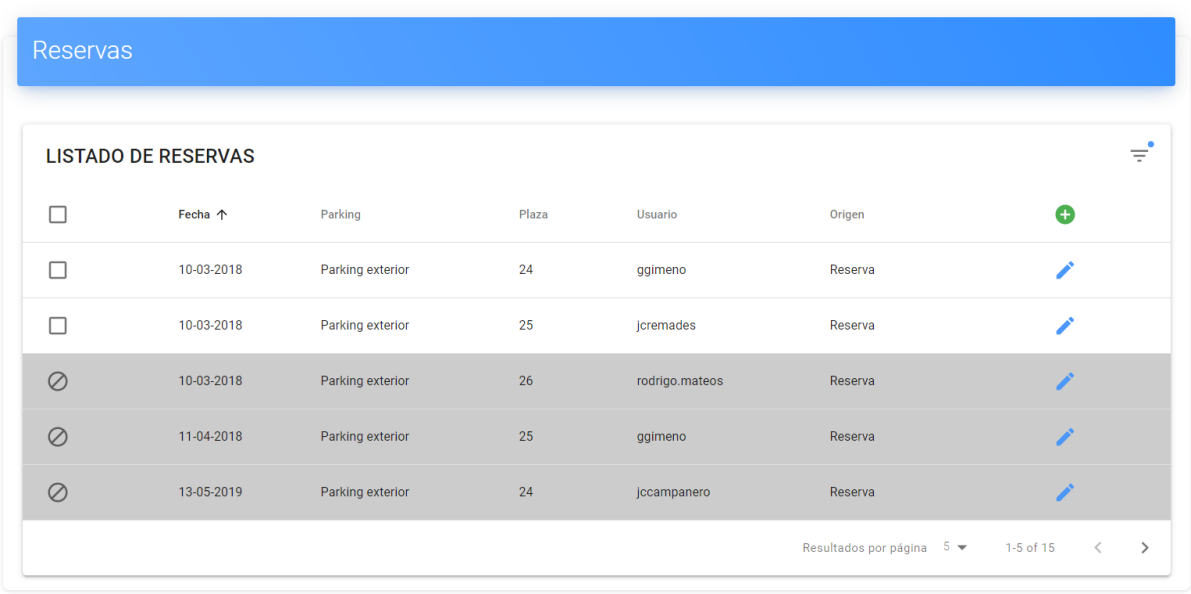

Figura 6-13. Captura de pantalla de IU: Listado de reservas

Como podemos ver en la *Figura 6-13*, la función para ver las reservas de un usuario es similar a la disponible para otras tablas de la aplicación, con la diferencia de que añade un filtrado con los parámetros:

- Fecha desde
- Fecha hasta
- Parking
- Plaza de parking
- Usuario
- Origen
- Mostrar canceladas

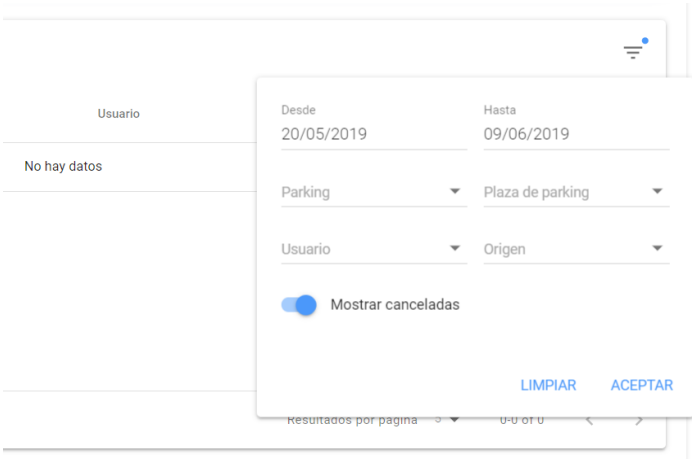

Figura 6-14. Captura de pantalla de IU: Filtrado del listado de reservas

Las reservas canceladas y la forma en la que el usuario administrador puede seleccionar varias reservas para cancelarlas masivamente es análoga a la gestión de plazas de parking (ver *Figura 6-7*).

Del mismo modo que en las otras pantallas de gestión, el usuario administrador podrá acceder al detalle de una reserva. La única diferencia es que no es posible modificar una reserva, sino que sólo tendrá las opciones de ver el detalle, crear nueva reserva o cancelar una existente. La información a consultar es:

- Usuario.
- Fecha.
- Parking.
- Plaza de parking.
- Origen de la reserva. Hay tres posibles orígenes:
  - Pre reserva. Proviene de la pre reserva de un usuario de la app.
  - Reserva. Proviene de la reserva realizada por un usuario de la app o si el usuario administrador la crea directamente.
  - Solicitud. Proviene de la solicitud realizada por un usuario de la app.
- Si está cancelada.

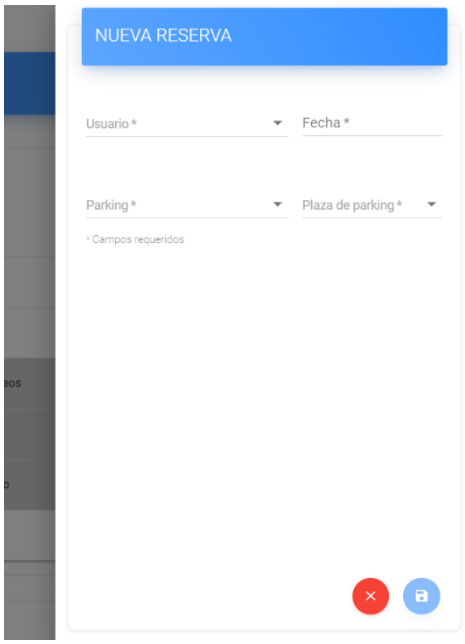

Figura 6-15. Captura de pantalla de IU: Formulario de reserva

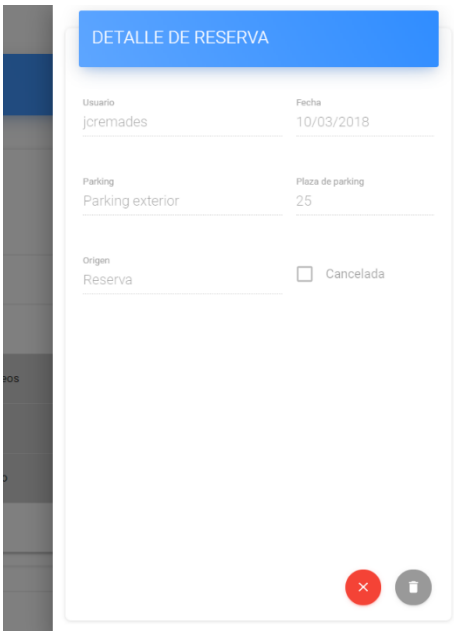

Figura 6-16. Captura de pantalla de IU: Detalle de reserva

Capítulo 7 - Arranque y despliegue de la aplicación

En este capítulo se describe cómo arrancar la aplicación web de administración y cómo se despliega una nueva versión en producción, además de explicar los diferentes perfiles que hay configurados para ello.

7.1 Perfiles de la aplicación

Una de las características de Spring es el uso de *Profiles*. Los *profiles* o perfiles nos permiten configurar grupos de elementos del framework para un "perfil" de ejecución predeterminado. La configuración de profiles nos permite cargar unas propiedades u otras y decidir si estamos en producción o desarrollo, en local o en cloud, etc.

7.1.1 Perfiles de *parking-core*

Actualmente, existen varios perfiles base de Spring:

- h2: configura la conexión a base de datos H2 en fichero.
- mysql: configura la conexión a base de datos MySQL en localhost.

Estos perfiles base se activan de forma agrupada por parte de los siguientes perfiles de Spring:

- local
- cloud

Por último, existe un perfil de Maven para cada perfil de Spring del listado anterior:

- local: perfil de Maven que activa el perfil 'local' de Spring.
- cloud: perfil de Maven que activa el perfil 'cloud' de Spring.

parking-core no se va a arrancar como servicio, es solo una librería de clases. A pesar de ello, al tener Liquibase, es necesario poder arrancar este proyecto para que actúe Liquibase y genere los cambios en el *changelog*. Por ello, es necesario también que este proyecto conecte con diferentes bases de datos, para hacer esta comparación. Ese es el motivo por el que hemos creado los perfiles que acabamos de explicar.

7.1.2 Perfiles de *parking-admin*

Actualmente, existen varios perfiles base de Spring:

- h2: configura la conexión a base de datos H2 en fichero.
- mysql: configura la conexión a base de datos MySQL en localhost. Para ello, podemos arrancar la base de datos con Docker.
- cloudsql: configura la conexión a base de datos en el servicio Cloud SQL de Google Cloud Platform.
- noauth: desactiva la autenticación, fijando un usuario de sesión, configurado en el archivo *application-noauth.yml*.

- azureauth: activa la autenticación con Azure.
- fcmnotifications: activa el envío de notificaciones push mediante Firebase.

Estos perfiles base se activan de forma agrupada por parte de los siguientes perfiles de Spring:

- local: activa los perfiles 'h2', 'noauth'. Se pueden modificar los perfiles a activar editando el fichero *application-local.yml*, para arrancar por ejemplo con una base de datos MySQL en vez de la H2 (para ello habría que quitar el perfil 'h2', y añadir el perfil 'mysql').
- cloud: activa los perfiles 'cloudsql', 'noauth' y 'fcmnotifications'.
- test: perfil de test.
- integration-test: perfil para ejecutar los test de integración.

Por último, existe un perfil de Maven para cada perfil de Spring del listado anterior:

- local: perfil de Maven que activa el perfil 'local' de Spring.
- cloud: perfil de Maven que activa el perfil 'cloud' de Spring.
- test: perfil de Maven que activa el perfil 'test' de Spring.
- integration-test: perfil de Maven que activa el perfil 'integration-test' de Spring.

7.2 Arranque en entorno local

En esta sección se indica los pasos a seguir para poder arrancar la aplicación en un entorno de desarrollo local.

7.2.1 Backend

7.2.1.1 Spring Boot

Para arrancar la aplicación desde local con Spring Boot, podemos hacer uso de Boot *Dashboard* del STS o de la línea de comandos con Maven:

```
mvn spring-boot:run -P<PERFIL_MAVEN>
```

Siendo PERFIL_MAVEN cualquiera de los perfiles de Maven explicados anteriormente en la sección *7.1 Perfiles de la aplicación*.

7.2.1.2 Proxy de Cloud SQL

Para arrancar el proxy de Cloud SQL en local, necesitamos tener el fichero JSON con las credenciales de la cuenta de servicio de tipo *project owner* creada en el proyecto de Google Cloud Platform.

- Creamos la variable de entorno GOOGLE_APPLICATION_CREDENTIALS apuntando a esta cuenta de servicio:

```
export GOOGLE_APPLICATION_CREDENTIALS="<PATH>/parking-
product-090bbcbcc0f9.json"
```

- Iniciamos el proxy, indicando la conexión a la base de datos en Cloud SQL:

```
   ./cloud_sql_proxy -instances=parking-database:europe-west1-
b:parking=tcp:3306
```

Con esto, podemos arrancar la aplicación en perfil *cloud*, y podremos tener la aplicación arrancada en local, pero apuntando a la base de datos desplegada en Cloud SQL.

7.2.2 Frontend

Desde el directorio donde se encuentra el código de la aplicación *parking-admin-frontend* ejecutamos:

```
    yarn add
```

Este comando descargará todas las dependencias del proyecto necesarias para arrancar la aplicación en la carpeta *node_modules.*

Para arrancar la aplicación en el navegador web, ejecutamos el comando:

```
    yarn start
```

Simplemente ejecutando estos dos comandos ya tendríamos arrancado en local el frontend de la aplicación de administración.

7.3 Despliegue en producción

7.3.1 Configuración de Kubernetes

El código para definir nuestra infraestructura de k8s lo podemos ver dentro del paquete *parking-admin* en los archivos de la carpeta */k8s/gke/:*

- ***parking-admin-deployment.yaml***

```
 1  apiVersion: apps/v1
 2  kind: Deployment
 3  metadata:
 4    name: parking-admin
 5  spec:
 6    selector:
 7      matchLabels:
 8        app: parking-admin
 9    replicas: 1
10    template:
```

```
11      metadata:
12        labels:
13          app: parking-admin
14      spec:
15        containers:
16          - name: parking-admin
17            image: gcr.io/parking-product/parking-admin:1.0.11
18            ports:
19              - containerPort: 8080
20            env:
21              - name: DB_HOST
22                value: 127.0.0.1:3306
23              - name: DB_USER
24                valueFrom:
25                  secretKeyRef:
26                    name: cloudsql-db-credentials
27                    key: username
28              - name: DB_PASSWORD
29                valueFrom:
30                  secretKeyRef:
31                    name: cloudsql-db-credentials
32                    key: password
33          - name: cloudsql-proxy
34            image: gcr.io/cloudsql-docker/gce-proxy:1.11
35            command: ["/cloud_sql_proxy",
36     "-instances=parking-product:europe-west1:parking-
database=tcp:3306",
37     "-credential_file=/secrets/cloudsql/credentials.json"]
38              securityContext:
39                runAsUser: 2  # non-root user
40                allowPrivilegeEscalation: false
41              volumeMounts:
42                - name: cloudsql-instance-credentials
43                  mountPath: /secrets/cloudsql
44                  readOnly: true
45        volumes:
46          - name: cloudsql-instance-credentials
47            secret:
48              secretName: cloudsql-instance-credentials
```

En este fichero se define el *deployment* del backend de *parking-admin*. En la línea 8 se define la *label* para que luego se pueda seleccionar el *pod* generado desde el *service*. La *label* es “app: parking-admin”.

En la línea 9 se especifica que el número de réplicas de este *deployment* es una. En principio, parece poco probable que sea necesario escalar esta instancia, ya que el uso de la web de administración no va a ser intensivo.

Después, en las líneas 16 y 33 se definen los contenedores Docker que van a estar desplegados en este deployment. Tenemos por un lado el contenedor “parking-admin”, que se genera a partir de la imagen de Docker generada para el backend de la aplicación, y, por otro lado, un contenedor llamado “cloudsql-proxy”, que sirve de proxy para conectar la aplicación a la base de datos, permitiendo una conexión segura [19].

Después, el usuario y contraseña de conexión a la base de datos se almacena como un secreto de Kubernetes, como se ve en las líneas 47 y 48. Un secreto o *secret* es un objeto de Kubernetes que nos permite almacenar datos confidenciales, como contraseñas o configuración.

- ***parking-admin-service.yaml***

```
 1  kind: Service
 2  apiVersion: v1
 3  metadata:
 4    name: parking-admin-service
 5  spec:
 6    selector:
 7      app: parking-admin
 8    ports:
 9    - name: http
10      protocol: TCP
11      port: 8080
12      targetPort: 8080
13    type: NodePort
```

Aquí se define el servicio que va a redirigir las peticiones al *pod* desplegado con el anterior *deployment*.

En la línea 7 se puede ver que este servicio seleccionará *pods* con la *label* “app: parking-admin”, lo cual coincide con lo definido en el anterior *deployment*.

7.3.2 Nueva versión en Google Kubernetes Engine

Los pasos a seguir para desplegar una nueva versión de *parking-admin* en Google Kubernetes Engine (GKE) se explican a continuación.

Vamos a partir del supuesto de que en GKE se encuentra ya desplegada una versión 1.0.0, y queremos desplegar una versión 1.1.0. Contamos por tanto con que ya hay desplegado en GKE un *deployment* y un *service*.

- Primero actualizaríamos la versión del proyecto:
  - Actualizamos versión del fichero pom a la 1.1.0
  - Actualizamos versión de la imagen Docker en el archivo *k8s/gke/parking-admin-deployment.yaml* a la 1.1.0.
- Empaquetamos la nueva versión de la aplicación:

```
mvn clean package -Pcloud
```

Cuando ejecutamos este comando, Maven empaqueta también *parking-admin-frontend* y lo sirve como recurso estático.

Al hacer el *package*, Maven crea también una nueva imagen de Docker: gcr.io/parking-product/parking-admin:1.1.0.

- Esta imagen ahora mismo está en nuestro repositorio local, y lo que tenemos que hacer es subirla al repositorio del Container Registry:

```
docker push gcr.io/parking-product/parking-admin:1.1.0
```

Una vez subida la imagen, podemos proceder a realizar el despliegue.

Como ya tenemos el *ingress* desplegado, no tenemos que volver a desplegarlo. Sin embargo, si tuviéramos que volver a desplegarlo, ejecutaríamos lo siguiente:

```
kubectl apply -f k8s/gke/parking-admin-ingress.yaml
```

De igual forma, como ya tenemos el *service* desplegado, no tenemos que volver a desplegarlo. Sin embargo, si tuviéramos que volver a desplegarlo, ejecutaríamos lo siguiente:

```
kubectl apply -f k8s/gke/parking-admin- service.yaml
```

Para el caso del *deployment*, tenemos que desplegarlo para que use la nueva imagen de Docker con la nueva versión.

Para desplegar la nueva versión del *deployment*, ejecutamos:

```
kubectl apply -f k8s/gke/parking-admin-deployment.yaml
```

De esta forma, Kubernetes debería comparar el *deployment* actual con el nuevo, y aplicar los cambios que existan (la nueva versión de la imagen Docker de la aplicación).

Capítulo 8 - Conclusiones

La aplicación que se ha implementado en este TFG se ha desarrollado satisfactoriamente, cumpliendo los requisitos especificados, y está lista para integrarse en con el resto del sistema y desplegarse en producción en Google Cloud Platform para ser utilizada, en primer lugar, por los empleados de CRC.

Al ser una aplicación relativamente fácil de utilizar, se requiere poco tiempo en aprender a realizar la gestión de los aparcamientos de empresas a través de ella, por lo que cualquier usuario estaría capacitado para usarla.

Además, el software desarrollado es mantenible y escalable, pudiendo realizar evolutivos de la aplicación con un esfuerzo relativamente bajo.

La interfaz de usuario que se ha desarrollado con React y Redux proporciona a los usuarios administradores las pantallas necesarias para gestionar las plazas del parking, gestionar los usuarios de la aplicación móvil y gestionar las reservas que se realizan. La interfaz de usuario obtiene los datos de la API que hemos desarrollado utilizando Spring Framework, que nos proporciona múltiples utilidades como inyección de dependencias, inversión de control o gestión de transacciones, entre otras.

Además, gracias a los distintos perfiles que tenemos disponibles en la aplicación, nos ofrece una forma fácil y rápida de cambiar de entorno de desarrollo a entorno de producción y de entorno *cloud* a entorno local.

Gracias al desarrollo de la aplicación web de administración he aprendido la importancia de tener una aplicación con alta disponibilidad. Esto es posible gracias a Kubernetes, que se encarga de mantener el “estado deseado” que hemos definido en los archivos de *deployment.* En estos archivos se indican ciertos contenedores que queremos tener desplegados y Kubernetes se encarga de que siempre estén desplegadas las instancias que hemos definido.

8.1 Trabajo futuro

Mientras se realizaba el desarrollo de la aplicación, han ido apareciendo ideas y futuras funcionalidades que se podrían incluir en la aplicación, ampliando la utilidad y facilitando la gestión del aparcamiento al usuario administrador.

Se han considerado las siguientes funcionalidades a desarrollar para las siguientes versiones de la aplicación:

- Gestión de usuarios administradores. Actualmente no existe una gestión desde la interfaz de usuario en la que se puedan consultar, añadir, modificar o dar de baja usuarios administradores.

- Carga masiva de ausencias. También se ha considerado de mucha utilidad desarrollar un *endpoint* donde los clientes puedan añadir masivamente todas las ausencias de sus empleados.

- Carga masiva de empleados. Similar al anterior punto es esta funcionalidad, pero en este caso el objetivo sería habilitar una API para cargar masivamente los empleados de una empresa.

- Gestionar configuración del Sorteo. Otro punto a implementar sería poder configurar el tipo de sorteo y la periodicidad con la que se ejecuta. Así, se podría configurar un sorteo que, por ejemplo, se ejecute una vez por semana, sin tener en cuenta la prioridad de los usuarios.

8.2 Valoración personal

El desarrollo de este TFG me ha aportado un punto de vista más realista del que acostumbramos a tener en la vida de estudiante, en el sentido de que en la vida real no se dan las cosas tal y como los problemas que planteamos y resolvemos en clase, ya que normalmente tienen una solución predefinida y estudiada. Por el contrario, en la vida real surgen problemas como los de este proyecto, cuya solución se desconoce desde un principio, así como el motivo por el que se producen, y nuestro cometido como ingenieros es aprender a resolverlos.

Por otra parte, este proyecto me ha servido para aprender una gran cantidad de conceptos de programación y desarrollo y metodologías de trabajo que antes desconocía. Unos conocimientos que complementan sin ninguna duda aquellos adquiridos en el transcurso de los últimos cuatro años en los que he estado estudiando en la Facultad de Informática de la UCM.

Chapter 8 - Conclusions

The application that has been implemented in this Final Degree Project has been developed satisfactorily, fulfilling the specified requirements, and is ready to be integrated with the rest of the system and be deployed in production in Google Cloud Platform to be used, in the first place, by the employees of CRC.

Being a relatively easy to use application, it takes little time to learn how to manage the car parks of companies through it, so any user would be able to use it.

In addition, the developed software is maintainable and scalable, being able to make evolutionary of the application with a relatively low effort.

The user interface that has been developed with React and Redux provides to user administrators the necessary screens to manage parking spaces, manage the users of the mobile application and manage the reservations. The user interface obtains the API data that we have developed using the Spring Framework, which provides us with multiple utilities such as dependency injection, control of inversion or transaction management, among others.

In addition, thanks to the different profiles that we have available in the application, it offers an easy and quick way to change from development environment to production environment and from the cloud environment to the local environment.

Thanks to the development of the administration web application I have learned the importance of having an application with high availability. This is possible thanks to Kubernetes, which is responsible for maintaining the "desired state" that we have defined in the deployment files. These files indicate certain containers that we want to have deployed and Kubernetes ensures that the instances we have defined are always deployed.

8.1 Future work

While the development of the application was carried out, some ideas and future functionalities that could be included in the application have been appearing, increasing the utility and facilitating the parking management to the administrator user.

The following features have been considered to be developed for the following versions of the application:

- Administrator user management. Currently there is no management from the user interface in which administrator users can be added, updated or unsubscribed or view their data.

- Load multiple absences at the same time. It has also been considered very useful to develop an endpoint where clients can massively add all the absences of their employees.

- Load multiple employees at the same time. Similar to the previous point is this functionality, but in this case the objective would be to enable an API to massively load the employees.

- Manage Raffle configuration. Another point to implement would be to configure the raffle type and the periodicity with which it is executed. Thus, you could set

up a raffle, for example, run once a week, without taking into account the users priority.

-

8.2 Personal assessment

The development of this Final Degree Project has given to me a more realistic point of view than we usually have in the student's life, in the meaning that in real life things do not happen as the problems we solve in class, since they usually have a predefined and studied solution. By contrast, in real life there are problems like those of this project, whose solution is unknown at the beginning, as well as the reason why they are produced, and our role as engineers is to learn how to solve them.

On the other hand, this project has helped to me in learning a great amount of programming and development concepts and work methodologies that I did not know before. This knowledge complements, without any doubt, the one that I have acquired during the last four years in which I have been studying in the Computer Science Faculty of UCM.

Bibliografía

[1] CRC Information Technologies, (2019). Disponible en http://www.crcit.es

[2] Schwaber, K., (2004). *Agile Project Management with Scrum*. Microsoft Professional

[3] +D3. *4Park Office*, (2019). Disponible en https://imasdetres.com/control-accesos-parking-oficinas-empresas/

[4] Urbiotica. *Sistema de guiado para aparcamientos de empresas y centros logísticos*, (2019). Disponible en https://www.urbiotica.com/soluciones-inteligentes-3/estacionamiento-guiado-en-empresas-y-centros-logisticos/

[5] Equinsa Parking. *Sistema de Control y gestión de aparcamiento: Sense*, (2019). Disponible en https://equinsaparking.com/soluciones-de-gestion/sistema-de-control-y-gestion-de-aparcamientos/

[6] SKIDATA. *Gestión de aparcamientos*, (2019), https://www.skidata.com/es-es/gestion-de-aparcamientos/

[7] Raona. (2017). *¿App nativa, web o híbrida?* Disponible en https://www.raona.com/aplicacion-nativa-web-hibrida/

[8] Lenguaje unificado de modelado. En *Wikipedia*. Recuperado el 24 de marzo de 2019 de https://es.wikipedia.org/wiki/Lenguaje_unificado_de_modelado

[9] Google Cloud Platform, (2019). Disponible en https://cloud.google.com

[10] Kubernetes. En *Wikipedia*. Recuperado el 15 de abril de 2019 de https://es.wikipedia.org/wiki/Kubernetes

[11] LearnITGuide.net. (2018). *What is Kubernetes – Learn Kubernetes from Basics*. Disponible en https://www.learnitguide.net/2018/08/what-is-kubernetes-learn-kubernetes.html

[12] Poulton, N. & Joglekar, P., (2019). *The Kubernetes Book.*

[13] Docker, (2019). Disponible en https://docs.docker.com/

[14] Spring by Pivotal Software, Inc. *Building Java Projects with Maven*, (2019). Disponible en https://spring.io/guides/gs/maven/

[15] Liquibase by Datical, (2019). Disponible en https://www.liquibase.org/documentation/index.html

[16] Bauer, C; King, G & Gregory, G., (2015). *Java Persistence with Hibernate (Second edition)*. Manning

[17] Hibernate ORM, Envers, (2019). Disponible en https://hibernate.org/orm/envers/

[18] Tutorialspoint. *Spring Tutorial*, (2019). Disponible en https://www.tutorialspoint.com/spring

[19] Google Cloud. *About the Cloud SQL Proxy*, (2019). Disponible en https://cloud.google.com/sql/docs/mysql/sql-proxy

[20] GeeksForGeeks. *Introduction to Apache Maven. A build automation tool for Java projects*, (2019). Disponible en *https://www.geeksforgeeks.org/introduction-apache-maven-build-automation-tool-java-projects/*

[21] Jenkins. *Jenkins User Documentation*, (2019). Disponible en https://jenkins.io/doc/

[22] *Material Desig*, (2019). Disponible en https://material.io/

[23] Hernández, J.M. (8 de septiembre, 2014). *ReactJS: un enfoque diferente*. Koalite. Disponible en http://blog.koalite.com/2014/09/reactjs-un-enfoque-diferente/

[24] Abramov, D. (23 de marzo, 2015). *Presentational and Container Components*. Disponible en https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

[25] React: *Getting Started*, (2019). Disponible en https://reactjs.org/docs/getting-started.html

[26] React, *Tutorial: Intro to React*, (2019). Disponible en https://reactjs.org/tutorial/tutorial.html

[27] Redux, (2019). Disonible en https://es.redux.js.org/

[28] Manjunath, M. (4 de mayo, 2018). *Getting Started With Redux: Why Redux?* Disponible en https://code.tutsplus.com/tutorials/getting-started-with-redux-why-redux--cms-30349

[29] *Pure function*. En *Wikipedia*. Recuperado el 3 de marzo de 2019 de https://en.wikipedia.org/wiki/Pure_function

[30] Redux, *Usage with React*, (2019). Disponible en https://redux.js.org/basics/usage-with-react

[31] Álvarez, MA. (19 de diciembre, 2014). Ventajas e inconvenientes de API REST para el desarrollo. Disponible en https://desarrolloweb.com/articulos/ventajas-inconvenientes-apirest-desarrollo.html

[32] *CRUD*. En *Wikipedia*. Recuperado el 17 de febrero de 2019 de https://es.wikipedia.org/wiki/CRUD

[33] Microsoft, *The Repository pattern*, (2019). Disponible en https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#the-repository-pattern

[34] Gierke, O., Darimont, T., Strobl, C., Paluch, M. & Bryant, J., (2019). *Spring Data JPA - Reference Documentation.* Disponible en https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

Apéndice A - Descripción detallada de las tablas en base de datos

- ***tpkn_absences***

Campo	Descripción
ID	Identificador único de la ausencia
DATE	Fecha de la ausencia
ID_USER	Clave externa para identificar el usuario

- ***tpkn_admin_users***

Campo	Descripción
ID	Identificador único del usuario administrador
LOGIN	*Username*
PASSWORD_HASH	Contraseña cifrada
FIRST_NAME	Nombre
LAST_NAME	Apellidos
EMAIL	Correo electrónico
ID_CLIENT	Clave externa para identificar el cliente al que pertenece

- ***tpkn_clients***

Campo	Descripción
ID	Identificador único del cliente
NAME	Nombre del cliente

- ***tpkn_parking_spots***

Campo	Descripción
ID	Identificador único de la plaza de parking
TITLE	Nombre de la plaza
ID_CLIENT	Clave externa para identificar el cliente al que pertenece
ID_PARKING	Clave externa para identificar el parking al que pertenece
ACTIVE	Indica si la plaza está activa. Sólo puede ser Y o N.

- ***tpkn_parkings***

Campo	Descripción
ID	Identificador único del parking
TITLE	Nombre del parking
ID_CLIENT	Clave externa para identificar el cliente al que pertenece
ACTIVE	Indica si la plaza está activa. Sólo puede ser Y o N.

- ***tpkn_preresevations***

Campo	Descripción
ID	Identificador único de la pre-reserva
FINAL_DATE	Fecha de fin de la pre reserva
INIT_DATE	Fecha de inicio de la pre reserva
ID_PARKING_SPOT	Clave externa para identificar la plaza de parking
ID_USER	Clave externa para identificar el usuario

- ***tpkn_raffle_execution_log***

Campo	Descripción
ID	Identificador único de la ejecución de sorteo
ID_RAFFLE	Clave externa para identificar el sorteo
ID_RAFFLE_EXEC_INIT	Si el registro es un *log* del final del sorteo, aquí se almacena el ID del registro del inicio de sorteo.
ID_RESOURCE	Dependiendo de la columna 'RAFFLE_LOG_TYPE', indica el ID del recurso (reserva, pre-reserva, etc), cuya creación, actualización o borrado ha provocado la ejecución del sorteo
LOG_DATE	Fecha del *log* de la ejecución de sorteo
LOG_TYPE	Tipo del *log* de la ejecución de sorteo. Si es un *log* del inicio del sorteo el valor es 0; si es de fin, 1
RAFFLE_LOG_TYPE	Indica la razón de la ejecución del sorteo. Puede tener los siguientes valores: RESERVATION_CANCELLED, PRERESERVATION_UPDATED, PRERESERVATION_DELETED, REQUEST_CREATED, JOB_EXECUTION, MANUAL_EXECUTION, OTHER_EXECUTION
SCHEDULED_DATE	Fecha planificada de la ejecución del sorteo. Podría no coincidir con la fecha real de la ejecución, por cualquier problema con el cron que ejecuta los sorteos.

- ***tpkn_raffles***

Campo	Descripción
ID	Identificador único del sorteo
CRON_EXP	Expresión que define la periodicidad en la que se debe ejecutar el proceso que realiza el sorteo
RAFFLE_TYPE	Tipo de sorteo. Puede tener los valores: RANDOM, RANDOM_WITH_PRIORITIES, RATIO o RATIO_WITH_PRIORITIES
ID_CLIENT	Clave externa para identificar el cliente al que pertenece

*Cron es un administrador de procesos en segundo plano que ejecuta procesos a intervalos regulares.

- ***tpkn_requests***: Es la tabla donde se almacenan las solicitudes realizadas por los usuarios de la aplicación.

Campo	Descripción
ID	Identificador único de la solicitud
DATE	Fecha de la solicitud
STATE	Estado de la solicitud. Puede ser ACTIVE o CANCELLED
ID_USER	Clave externa para identificar el usuario que realiza la solicitud

- ***tpkn_reservations***

Campo	Descripción
ID	Identificador único de la reserva
DATE	Fecha de la reserva
ID_ORIGIN	Hace referencia al ID de la pre-reserva o solicitud que ha originado la reserva. Si la reserva se ha creado directamente, entonces este campo estará vacío.
ORIGIN	Origen de la reserva. Puede ser RES, PRE o REQ (Si el origen es por un Reserva, una Pre-reserva o una Solicitud, respectivamente)
STATE	Estado de la reserva. Puede ser ACTIVE o CANCELLED
ID_PARKING_SPOT	Clave externa para identificar la plaza de parking
ID_USER	Clave externa para identificar el usuario

- ***tpkn_roles***

Campo	Descripción
ID	Identificador único del rol
ROLE	Rol del usuario
ID_USER	Clave externa para identificar el usuario

- ***tpkn_users_app***

Campo	**Descripción**
ID	Identificador único del usuario de la aplicación
ABSENCE_BEHAVIOUR	Comportamiento en caso de que el usuario tenga una ausencia y se le asigne una plaza. Puede ser MAINTAIN o RELEASE
NAME	Nombre del usuario
PRIORITY	Prioridad utilizada al realizar el sorteo. Puede ser 1, 2 o 3
SURNAME	Apellidos
USERNAME	*Username*
ID_CLIENT	Clave externa para identificar el cliente al que pertenece
NOTIFICATIONS	Indica si el usuario recibirá notificaciones si se le asigna una plaza en el sorteo. Sólo puede ser Y o N.
ACTIVE	Indica si el usuario está activo. Sólo puede ser Y o N.